Modeling and Verification of Sigma Delta Neural Networks

A dissertation submitted in partial fulfilment for the degree of

Master of Technology

in

Computer Science

by

Sirshendu Das

Roll No. CS2323

under the supervision of

Dr. Ansuman Banerjee

Professor, Advanced Computing and Microelectronics Unit (ACMU)

and

Dr. Swarup Kumar Mohalik

Principal Researcher, Ericsson Research, Bangalore

Indian Statistical Institute, Kolkata

June, 2025

Dedicated to my beloved parents
and my teacher Dr. Debayan Ganguly.

Certificate

This is to certify that the dissertation titled **"Modeling and Verification of Sigma Delta Neural Networks"**, submitted by **Sirshendu Das** to the Indian Statistical Institute Kolkata, for the award of the degree of **Master of Technology in Computer Science**, is a record of original, bonafide research work carried out by him under our supervision and guidance. The dissertation has reached the standards fulfilling the requirements of the regulations related to the award of the degree. The results contained in this dissertation have not been submitted in part or in full to any other University or Institute for the award of any degree or diploma to the best of our knowledge.

.....

Dr. Ansuman Banerjee

Advanced Computing and Microelectronics Unit (ACMU),

Indian Statistical Institute, Kolkata.

.....

Dr. Swarup Kumar Mohalik

Ericsson Research,

Bangalore.

Declaration

I, **Sirshendu Das**, with Roll No. **CS2323**, hereby declare that the material presented in the dissertation titled **"Modeling and Verification of Sigma Delta Neural Networks"** represents original work carried out by me for the degree of **Master of Technology in Computer Science** at the **Indian Statistical Institute, Kolkata**.

Furthermore, I affirm that no sections of this report have been sourced or copied from external references without proper attribution. I am aware that any instances of plagiarism or the use of unacknowledged materials from third parties will be treated with the utmost seriousness and consequences.

Sirshendu Das

M.Tech (CS), CS2323

Indian Statistical Institute

Acknowledgement

I extend my sincere appreciation to **Dr. Ansuman Banerjee**, my advisor at the Advanced Computing and Microelectronics Unit (ACMU) of Indian Statistical Institute Kolkata, for his guidance, continuous support, and inspiration. His profound knowledge and creative suggestions have taught me a great deal in every subject and have shown me how to conduct solid research.

I would like to sincerely thank **Dr. Swarup Kumar Mohalik**, my co-supervisor at Ericsson Research, Bangalore, for his invaluable guidance for this research. His consistent provision of ideas and unwavering support have been instrumental for the success of this project. I acknowledge the support of Ericsson Research for partial support towards my dissertation.

I am sincerely grateful to the faculty of the Indian Statistical Institute for their valuable guidance and instruction, which significantly shaped my research perspective. I especially thank **Sruti Goswami** and the seniors at ACMU for their continued mentorship. I also extend my heartfelt thanks to **Dr. Debayan Ganguly** and **Dr. Kingshuk Chatterjee** for their insightful feedback and constant support.

Finally, I want to express my gratitude to my parents and extended family for their unwavering support. I also extend my sincere appreciation to all my friends for their continuous assistance and encouragement. I am thankful to everyone who has contributed to my growth and success, even if I have inadvertently missed mentioning them in the above list.

Abstract

In the context of modern day embedded safety-critical systems and low-resource edge devices in particular, Sigma-Delta Neural Networks (SDNNs) offer a promising alternative to traditional Artificial Neural Networks (ANNs) by leveraging event-driven, sparse computations inspired by biological neural processing. This energy-efficient paradigm makes SDNNs well-suited for neuromorphic hardware and real-time applications, particularly in scenarios with temporal redundancy, such as video processing. However, as neural networks become integral to safety-critical systems, ensuring their robustness against adversarial perturbations is an absolute necessity. In this work, we propose an end-to-end framework for formal modeling and verification of SDNNs using Satisfiability Modulo Theory (SMT). Unlike empirical robustness evaluations, SMT-based verification provides formal guarantees by encoding SDNN behavior and adversarial robustness properties as mathematical constraints. We introduce an SMT-based formulation for encoding SDNNs with SMT constraints and define a robustness property motivated by video stream processing. Our approach systematically examines how well SDNNs can handle adversarial attacks, ensuring they work correctly in safety-critical applications. We validate our framework through experiments on temporal version of the MNIST dataset. To the best of our knowledge, this is the first formal verification framework for SDNNs, bridging the gap between neuromorphic computing and rigorous verification. We also focus on applying the proposed SDNN verification methodology to a real-world deep learning system– PilotNet, an end-to-end model for steering angle prediction in autonomous vehicles.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent times, Neural networks are being extensively used in safety critical embedded systems such as autonomous vehicles [1] and airborne collision avoidance systems [2]. This has mandated the need for rigorous methods to ensure correct functioning of neural network controlled systems. Formal verification offers a great promise in that direction by being able to derive strong guarantees of correctness by exhibiting a proof. Formal verification of neural networks has been studied actively in recent years. [3, 4, 5, 6, 7].

Traditional deep networks rely on dense computations, leading to high energy consumption. A promising alternative is Spiking Neural Networks (SNNs), inspired by the brain's event-driven processing, which operate asynchronously and can significantly reduce power consumption [8]. These energy-efficient models hold potential for neuromorphic hardware and embedded real-time edge applications. The need for ensuring their smooth and trustworthy integration into the embedded system design flow has inspired research on safety verification and bound analysis for SNN [9, 10, 11].

In this dissertation, we first explore Sigma-Delta Neural Networks (SDNNs), a variant of SNNs, that is inspired by principles of efficient communication and sparsity [12]. Much like how our brains process only the most critical changes in sensory

inputs, SDNNs compute only the differences—quantized, discrete changes—between consecutive states. This revolutionary approach reduces computational burdens, making SDNNs a compelling alternative to traditional deep learning frameworks for scenarios with temporal redundancy, such as video processing. By focusing on changes rather than redundancies, SDNNs promise a path towards energy-efficient AI without compromising performance. As demonstrated by their application to tasks such as the Temporal-MNIST classification [12], SDNNs can process data efficiently while preserving accuracy, unlocking a new frontier for sustainable AI for safety critical embedded systems.

As SDNNs get widely adopted in the embedded safety critical systems context [13], the question of their trustworthiness and correctness becomes pertinent. This is what we aim to address in this dissertation. In particular, we address the question of robustness of SDNNs against adversarial attacks. As shown in the context of Artificial Neural Networks (ANNs), subtle yet carefully crafted perturbations can cause neural networks to mispredict, undermining their reliability in critical applications [14, 15, 16]. Verifying the robustness of such systems is no longer a luxury but a necessity, particularly for networks designed for safety-critical environments. Neural networks, as universal function approximators [17], must be rigorously analyzed to estimate their robustness. Significant progress has been made on formal verification of traditional ANNs [4, 5, 6, 7], but the integration of such techniques with SDNNs remains a challenge. The intersection of efficiency, correctness and robustness to perturbation forms the crux of our exploration into formal methods-based modeling and verification of SDNNs—a journey to unlock their full potential while safeguarding their deployment. Our main contribution is an end-to-end framework for modeling and verification of SDNNs against adversarial attacks using SMT constraints.

Satisfiability Modulo Theory (SMT) is a powerful modeling formalism for formal verification that extends Boolean satisfiability (SAT) by incorporating constraints

from various mathematical theories such as linear arithmetic, real numbers, and bit-vectors [18]. SMT solvers efficiently determine whether a given formula defined over a theory is satisfiable, making them a valuable tool for verifying the correctness and robustness of complex systems, including neural networks. In the context of SDNNs, SMT-based verification offers a principled approach to ensure their reliability against adversarial perturbations. Unlike heuristic robustness evaluations, which often rely on empirical testing, SMT provides formal guarantees by exhaustively analyzing the network's behavior within defined constraints. Given SDNNs' unique event-driven processing and temporal dependencies, classical verification methods developed for ANNs are not directly applicable. On the other hand, SMT with its capability to encode discrete state transitions and sparsity constraints emerges as a compelling framework for analyzing SDNNs against adversarial and safety-critical conditions. While SMT encodings for SNNs [11] can model temporal dynamics of signals, novel neuron types of SDNN require new SMT encoding tailored to the unique operational semantics of these neurons. By leveraging SMT, we aim to bridge the gap between energy-efficient neuromorphic computing and formal verification approaches for deriving robustness guarantees, ensuring that SDNNs can be deployed with confidence in real-world applications.

This dissertation brings together two core contributions in the realm of SDNN verification, as presented in our recent research. The first contribution presents a symbolic modeling and verification framework for SDNNs, utilizing SMT to encode their dynamics and formally reason about their behavior under adversarial perturbations. Unlike traditional empirical testing methods, our SMT-based verification approach offers mathematically rigorous guarantees, ensuring that the network adheres to desired specifications even in the presence of input perturbations. To this end, we define a novel robustness property tailored to temporal data and develop a scalable encoding of SDNN execution semantics. Our framework is validated through

experiments on the temporal version of the MNIST dataset.

Building on this foundational work, the second contribution focuses on applying the proposed SDNN verification methodology to a real-world SDNN-based deep learning system– PilotNet, an end-to-end model for steering angle prediction in autonomous vehicles [1, 13]. PilotNet traditionally relies on dense convolutional networks to process video frames captured by dashboard cameras. Given the temporal continuity inherent in driving scenarios, we use a re-engineered PilotNet as an SDNN-based model to exploit temporal sparsity and reduce redundant computation. We then develop an abstraction-refinement-based SMT verification framework to model the symbolic execution of this SDNN-based PilotNet. This instantiation not only demonstrates the practical relevance of our general framework but also highlights how domain-specific architectures can benefit from SDNN-based reformulation.

To the best of our knowledge, this dissertation presents the first unified approach for symbolic modeling and robustness verification of SDNNs, including their integration into large-scale vision-based models like PilotNet. By bridging the gap between neuromorphic efficiency and formal correctness, our contributions pave the way for safe and reliable deployment of energy-efficient neural networks in real-world systems.

The structure of the dissertation is as follows: Chapter 2 surveys related work in formal verification of neural networks. Chapter 3 introduces Sigma Delta Neural Networks, its architecture and temporal processing capabilities. Chapter 4 details our symbolic SMT encoding of SDNNs, laying the foundation for verification. Chapter 6 describes the SDNN-based PilotNet system, explaining their architecture and temporal processing capabilities. Chapter 7 explains the verification of PilotNet using the abstraction-refinement framework. Chapter 5 presents experimental results and verification outcomes on benchmark datasets. Chapter 8 concludes this dissertation.

Chapter 2

Related Work

In this chapter, we highlight some relevant works in the domain of modeling and verification of traditional ANNs as well as Convolutional Neural Networks (CNNs) and SNNs. Also, we explain Satisfiability Modulo Theory (SMT), which is required as a preliminary methodology for this dissertation.

2.1 Formal Verification of Neural Networks

Formal verification of neural networks has emerged as a vibrant research field, with a particular focus on ensuring robustness [4]. Constraint-based verification approaches rely on the principles of SMT to analyze the behavior of neural networks. This theoretical framework provides a structured way to reason about the properties of the network, ensuring rigorous evaluation and validation. [5] expanded the simplex method to enable the verification of neural networks with ReLU activation functions, which are widely used in deep learning architectures. Building on this work, [19] developed the Marabou framework by extending support to arbitrary piecewise linear activation functions, broadening its applicability in neural network verification. More recently, $\alpha, \beta-$CROWN [20, 21, 22] has emerged as a fast and scalable neural network

verifier with efficient bound propagation, significantly improving verification performance through optimized linear relaxation techniques. Constraint-based methods leverage SMT solvers, which are well-suited for handling discrete activations, making them a more adaptable choice for verifying SNNs and SDNNs. In recent years, SMT solvers are gaining widespread popularity because of their ability to handle a wide arsenal of theories.

2.2 Formal Verification of CNNs

While numerous techniques exist for the formal verification of general Deep Neural Networks (DNNs), verifying Convolutional Neural Networks (CNNs) introduces additional challenges due to their unique structural components like convolution and max-pooling layers. These layers significantly increase the complexity and scalability concerns when traditional verification methods designed for fully connected networks are applied.

Some work has explicitly targeted CNN verification, proposing reachability based approaches, either complete [23] or incomplete [24, 25, 26]. In [27], the authors proposed a specialized framework, `CNN-Abs`, which employs an abstraction-refinement approach tailored for CNNs. The method over-approximates the network by removing convolutional connections, creating a simpler fully connected abstraction. If the abstraction fails to verify the property, it incrementally refines the network by restoring previously removed neurons and connections. This approach allows existing verification backends (e.g., Marabou) to scale more efficiently, leveraging the reduced network size during the verification process.

2.3 Formal Verification of SNN

As SNNs gain attraction in modern times, it becomes crucial to ensure their robustness against adversarial attacks. Recognizing this challenge, [11] introduced an SMT-based encoding framework tailored for SNN verification. Their approach adapts quantifier-free linear real arithmetic constraints to capture the mathematical operations of SNNs. To assess adversarial robustness, they formulated logical constraints that account for potential perturbations in the input data. By leveraging SMT solvers, they demonstrated the feasibility of this method for robustness verification.

The mathematical models for SDNNs differ substantially from those for ANNs and SNNs due to their event-driven and time-varying nature. SDNNs operate on real-valued inputs, allowing them to capture more precise temporal and spatial information from the real world. This real-valued representation provides SDNNs with a unique advantage in applications requiring fine-grained signal processing, making them distinct from both conventional ANNs and third-generation SNNs. Despite the growing interest in neural network verification, most existing methods focus on ANNs and SNNs, leaving SDNNs largely unexplored. Given the complexity introduced by the Sigma-Delta mechanism, verifying the correctness and robustness of SDNNs requires new approaches. In this work, we introduce an SMT-based modeling framework specifically designed for SDNN verification, marking a significant step forward in the formal verification of event-driven neural networks. To the best of our knowledge, this is the first research effort for modeling SDNNs within an SMT-based framework, filling a critical gap in the field. Further, we address the question of adversarial robustness in the context of SDNNs with a completely new definition.

2.4 Satisfiability Modulo Theory (SMT)

Satisfiability Modulo Theory (SMT) is a decision problem that extends the classical Boolean satisfiability problem (SAT) by incorporating background theories such as linear arithmetic, bit-vectors, arrays, real numbers, and uninterpreted functions [18]. An SMT solver determines whether a logical formula is satisfiable under a given theory, making it a powerful tool for formal verification of complex systems. Unlike SAT solvers that operate over Boolean variables, SMT solvers reason about richer structures, enabling more expressive modeling of system behaviors.

In the context of neural networks, SMT has been successfully employed to verify properties such as robustness, safety, and fairness by encoding the network's operations and properties as logical constraints [28, 29]. Our work leverages the expressiveness of SMT to model the unique temporal dynamics of Sigma Delta Neural Networks (SDNNs), capturing both their quantized communication and temporal sparsity. By formulating adversarial robustness verification as an SMT satisfiability query, we obtain formal guarantees of correctness that go beyond empirical testing. Furthermore, the modularity of SMT encodings allows us to extend our framework to domain-specific models such as PilotNet, demonstrating the generality of SMT-based verification in neuromorphic architectures.

2.4.1 SMT-Based Verification of Neural Networks

Satisfiability Modulo Theories (SMT) solvers have been extensively used for formal verification of neural networks due to their ability to reason about arithmetic, logical, and structural constraints. Among the SMT solvers, Z3 [30] developed by Microsoft Research, has emerged as one of the most widely adopted tools in the domain of program verification and symbolic reasoning. Z3 has also been employed in the context of deep learning verification to reason about piecewise-linear activations,

such as ReLU, and input-output relations of neural models [29]. Frameworks like Reluplex, Marabou [31], and others internally rely on SMT-based reasoning or similar abstractions to establish formal properties.

While traditional SMT applications target static feedforward neural networks, recent work has begun exploring their applicability to temporal and spiking neural architectures. In this line of research, our work leverages Z3—specifically its Python interface Z3Py—for encoding and verifying Sigma-Delta Neural Networks (SDNNs).

Adversarial Robustness

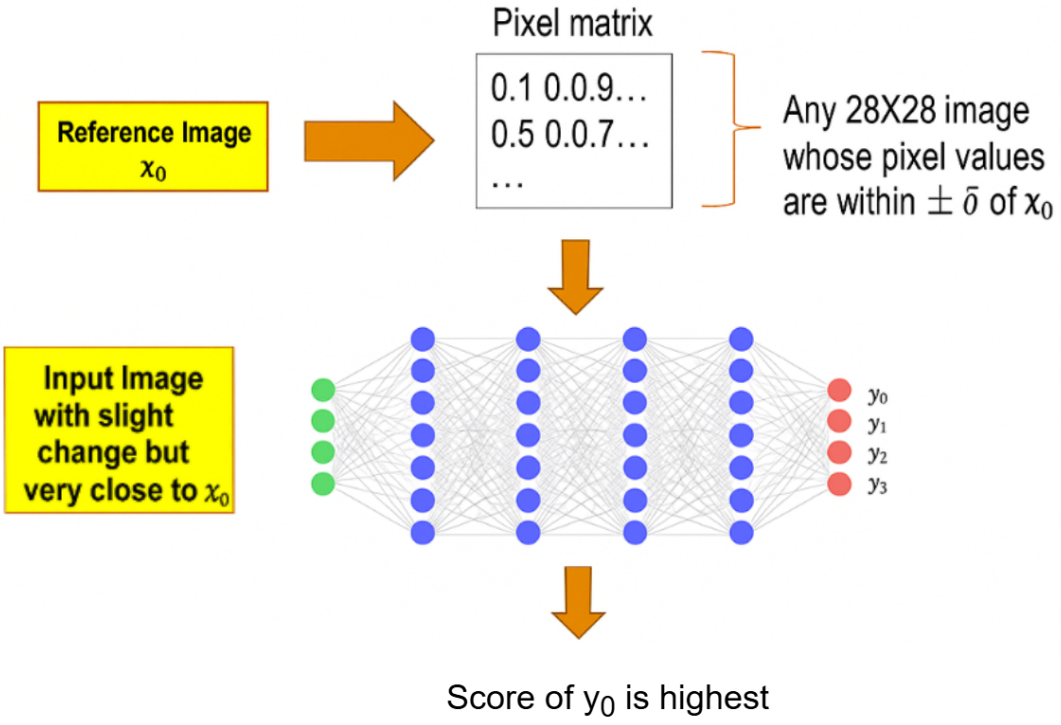

FIGURE 2.1: Adversarial Robustness example for an image x_0 of size 28×28 with label y_0

Adversarial robustness refers to a neural network's ability to maintain consistent predictions in the presence of small perturbations to its input. For image classification tasks, this means that if an image is altered slightly—such that each pixel deviates

by no more than a small value δ from a reference image x_0—the network should still assign the same label to the perturbed image. To formally verify this, we define the δ-neighbourhood of x_0 as:

$$\|x - x_0\|_\infty \leq \delta$$

This constraint ensures that all pixel values in the perturbed input x remain within $\pm\delta$ of the corresponding pixels in x_0.

Verification is performed using constraint solvers to check whether all such x within the δ-ball yield the same classification as x_0. The condition to be satisfied can be expressed as:

$$\bigvee_i (y[i_0] \leq y[i])$$

where $y[i_0]$ is the output corresponding to the correct label. If the solver returns `UNSAT`, it implies no input within the δ-ball leads to a misclassification, and thus, the network is adversarially robust for the given δ.

This setup provides a principled approach to verifying robustness and forms the foundation for more comprehensive robustness guarantees.

Local vs. Global Adversarial Robustness.

In SDNN verification, robustness is categorized as local or global. Local robustness ensures that small input perturbations do not alter the network's output, crucial for defending against adversarial attacks. If an SDNN is locally robust, its predictions remain unchanged within a small neighborhood of any input. Global robustness evaluates stability across the entire input space, ensuring reliable predictions for all possible inputs. Although more challenging to achieve, it guarantees broader model reliability. In SDNN verification, checking local robustness helps assess how well the model can withstand minor variations, whereas verifying global robustness provides a comprehensive guarantee of the network's reliability under all conditions. Both play

a crucial role in determining the trustworthiness and safety of SDNNs in real-world applications.

Chapter 3

Sigma Delta Neural Network

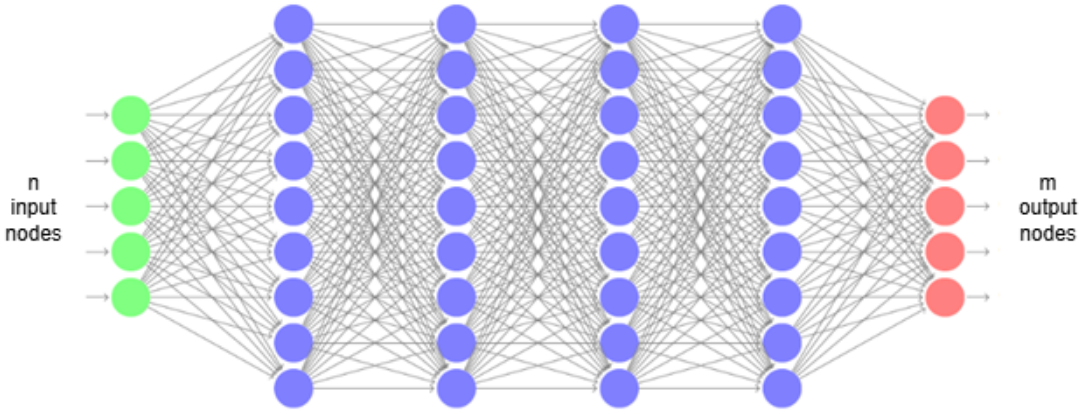

FIGURE 3.1: Sigma Delta Neural Network ($\mathcal{N}$)

A **Sigma Delta Neural Network (SDNN)** is a specialized type of neural network having an architecture similar to feedforward neural networks, however, differing in the semantic operation. An SDNN processes changes in activation between consecutive inputs rather than recomputing the entire activation. The architecture of an SDNN $\mathcal{N}$ depicted in Figure 3.1 consists of the following components:

- **Input Layer:** The network starts with n input neurons. Rather than feeding the raw input values, SDNNs operate on the temporal difference of inputs, i.e., $\Delta x_t = x_t - x_{t-1}$ where x_t and x_{t-1} denote the values of the input x at time

steps t and t-1 respectively.. This allows the network to focus on changes in input over time, enhancing computational efficiency.

- **Hidden Layers:** The network includes a number of fully connected hidden layers ((4 in the figure above) consisting of Sigma-Delta neurons. These layers propagate the delta information forward. Due to the sparsity of the deltas, many neurons may remain inactive unless there is a significant temporal change, reducing the number of computations required per timestep.

- **Output Layer:** The final layer comprises of m output neurons. These neurons perform the Sigma operation, which accumulates the incoming deltas from the last hidden layer to reconstruct the full output activations over time.

This architecture is particularly beneficial for processing time-varying data such as video streams or event-based sensor readings, where only a subset of the network needs to be activated at any given moment, thereby saving power and improving efficiency. This is achieved through two key mechanisms: **temporal difference** and **temporal integration**, which focus on detecting and accumulating changes in the input over time. Additionally, SDNNs use quantization to encode changes in input signals as discrete values, improving computational efficiency and enabling effective processing of temporal data. We first define the **temporal difference** (Δ_T) and **temporal integration** (Σ_T) modules in the procedures below.

Procedure Temporal Difference(Δ_T)

Internal: $\vec{x}_{\text{last}} \in \mathbb{R}^d \leftarrow \vec{0}$
Input: $\vec{x} \in \mathbb{R}^d$
Output: $\vec{y_\Delta} \in \mathbb{R}^d$
1: $\vec{y_\Delta} \leftarrow \vec{x} - \vec{x}_{\text{last}}$
2: $\vec{x}_{\text{last}} \leftarrow \vec{x}$
3: **return** $\vec{y_\Delta}$

Procedure Temporal Integration(Σ_T)

Internal: $\vec{y_\Sigma} \in \mathbb{R}^d \leftarrow \vec{0}$
Input: $\vec{x} \in \mathbb{R}^d$
Output: $\vec{y_\Sigma} \in \mathbb{R}^d$
1: $\vec{y_\Sigma} \leftarrow \vec{y_\Sigma} + \vec{x}$
2: **return** $\vec{y_\Sigma}$

Temporal data refers to data recorded or observed over time, where the order and time intervals between observations are important. Unlike static data, which represent a snapshot at a single point in time, temporal data capture how values evolve over a sequence of time steps. Temporal data can be processed efficiently using the Δ_T and Σ_T modules, capturing changes rather than raw values, making them energy efficient for time-dependent applications.

A SDNN consists of neurons that process temporal data. It begins with an input signal that is rounded to the nearest quantized level using the function Round(x). The rounded input is passed through the Δ_T module, which computes the temporal differences to capture changes over time. The output of Δ_T is weighted by $w(x)$ to model synaptic interactions. Subsequently, the weighted output is processed by the Σ_T module, which performs temporal integration to accumulate information over time. The integrated signal is passed through a nonlinear activation function $h(x)$, such as ReLU or sigmoid, to introduce nonlinearity and model complex patterns. Finally, the processed signal is output for the desired task. The final output is an aggregation of the output sequence (for example, *majority_voting* of the values over the sequence length). Figure 3.2 is a simple flowchart to understand different operations and modules of SDNN. An SDNN with multiple layers of nodes may use thresholding and residual states within the neurons. In this work, we do not consider thresholding and residual states.

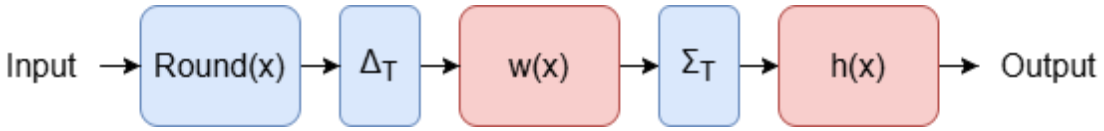

FIGURE 3.2: Working principle of one neuron in Sigma Delta Neural Network

To illustrate the processing of temporal data by a SDNN, we define a temporal dataset consisting of a sequence of inputs observed over a *temporal window* of length T. An SDNN processes an input $\vec{x} \in \mathbb{R}^{n \times T}$ and outputs $\vec{y} \in \mathbb{R}^{m \times T}$, where $\mathbb{R}$ denotes the set of real numbers and n, m are the number of inputs and outputs respectively. In the SDNN, each vector of size n at a given time step $t \in [1, \ldots, T]$ in the input layer is referred to as a *frame* in the input sequence.

Definition 3.1 (**Temporal Window**)**.** A temporal window is a fixed-length sequence of consecutive time steps extracted from temporal data, preserving the order and time intervals between observations. Each observation within a temporal window is known as a frame.

The following example explains the philosophy behind working of an SDNN on temporal data.

Example 1: Consider a simple SDNN with 3 input features, 1 hidden layer, and 1 output feature, shown in Figure 3.3. Let for input $\vec{x} \in \mathbb{R}^{3 \times T}$, that is, at each time step $t \in [1, T]$, the system receives a 3-input vector $\vec{x}_t = [x_1^t, x_2^t, x_3^t] \in \mathbb{R}^3$ consisting of three input features. The output at time step t is denoted as o_t. For simplicity of illustration and ease of explanation, we consider a simple activation function $h(x) = x$ and zero bias in each node in each layer. The SDNN processes the changes in these inputs using the Δ_T and Σ_T modules as described above. Table 3.1 presents an example sequence of input values over a temporal window of 5, demonstrating how SDNN captures temporal dynamics through the difference and integration mechanisms. Note for any SDNN, each input layer node has only the Δ_T module

and each output layer node has only the Σ_T module. All hidden layer nodes have both Δ_T and Σ_T modules as described in the procedures above.

Time(t)	**1**	**2**	**3**	**4**	**5**
x_1	1.2	3.2	3.3	2.9	4
x_2	0.2	1.1	0.9	-1	1
x_3	0.3	0	0.1	0	0
y_t	1	7	7	-1	8

TABLE 3.1: Sample temporal data with features x_1, x_2, x_3, and output y over a *temporal window* of 5.

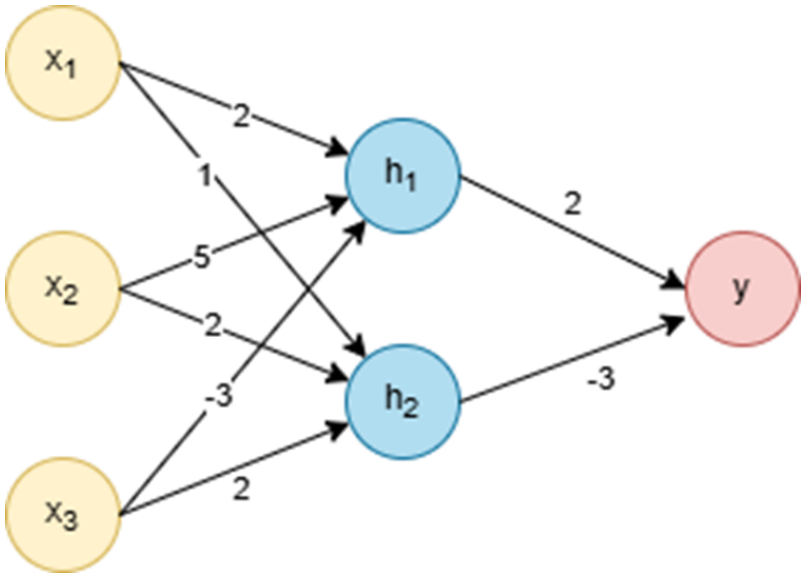

FIGURE 3.3: SDNN example

At $t = 0$, for all layers, we initialize $\vec{x}_{\text{last}} \leftarrow \vec{0}$, $\vec{y_\Sigma} \leftarrow \vec{0}$ corresponding to the Δ_T and Σ_T modules described above.

At $t = 1$, the input $\vec{x} = (1.2, 0.2, 0.3)$ is first rounded to $(1, 0, 0)$. Then Δ_T of the input layer returns $[(1, 0, 0) - (0, 0, 0)] = (1, 0, 0)$ and updates its $\vec{x}_{\text{last}} \leftarrow (1, 0, 0)$. It propagates through the corresponding weights. As a result Σ_T of the hidden layer gets $[(2 \cdot 1 + 5 \cdot 0 + (-3) \cdot 0), (1 \cdot 1 + 2 \cdot 0 + 2 \cdot 0)] = (2, 1)$ as input. Then Σ_T of the hidden layer returns $[(2, 1) + (0, 0)] = (2, 1)$ and updates its $\vec{y_\Sigma} \leftarrow (2, 1)$. After

Time(t)	**Input($\vec{x_t}$)**	**Round($\vec{x_t}$)**	$\vec{x}$	$\vec{x_{last}}$	$\vec{y_\Delta}$	**w($\vec{y_\Delta}$)**
1	(1.2,0.2,0.3)	(1, 0, 0)	(1, 0, 0)	(0, 0, 0)	(1, 0, 0)	(2, 1)
2	(3.2,1.1,0)	(3, 1, 0)	(3, 1, 0)	(1, 0, 0)	(2, 1, 0)	(9, 4)
3	(3.3,1.1,0.1)	(3, 1, 0)	(3, 1, 0)	(3, 1, 0)	(0, 0, 0)	(0, 0)
4	(2.9,-1,0)	(3, -1, 0)	(3, -1, 0)	(3, 1, 0)	(0, -2, 0)	(-10, -4)
5	(4,1,0)	(4,1,0)	(4,1,0)	(3, -1, 0)	(1, 2, 0)	(12, 5)

TABLE 3.2: Δ- Computations at the input layer

Time(t)	$\vec{x}$	$\vec{y_{\Sigma t-1}}$	$\vec{y_{\Sigma t}}$	**h($\vec{y_\Sigma}$)**	**Round($\vec{x_t}$)**	$\vec{x}$	$\vec{x_{last}}$	$\vec{y_\Delta}$	**w($\vec{y_\Delta}$)**
1	(2, 1)	(0, 0)	(2, 1)	(2, 1)	(2, 1)	(2, 1)	(0, 0)	(2, 1)	(1)
2	(9, 4)	(2, 1)	(11, 5)	(11, 5)	(11, 5)	(11, 5)	(2, 1)	(9, 4)	(6)
3	(0, 0)	(11, 5)	(11, 5)	(11, 5)	(11, 5)	(11, 5)	(11, 5)	(0, 0)	(0)
4	(-10, -4)	(11, 5)	(1, 1)	(1, 1)	(1, 1)	(1, 1)	(11, 5)	(-10, -4)	(-8)
5	(12, 5)	(1, 1)	(13, 6)	(13, 6)	(13, 6)	(13, 6)	(1, 1)	(12, 5)	(9)

TABLE 3.3: $\Sigma - \Delta$ Computations at the hidden layer

Time(t)	$\vec{x}$	$\vec{y_{\Sigma t-1}}$	$\vec{y_{\Sigma t}}$	**h($\vec{y_\Sigma}$)**	$\vec{y}$
1	(1)	(0)	(1)	(1)	(1)
2	(6)	(1)	(7)	(7)	(7)
3	(0)	(7)	(7)	(7)	(7)
4	(-8)	(7)	(-1)	(-1)	(-1)
5	(9)	(-1)	(8)	(8)	(8)

TABLE 3.4: Σ- Computations at output layer

passing through the activation function, we get $(h_1, h_2) = (2, 1)$. Then (h_1, h_2) after rounding remains (2, 1). Next, Δ_T of the hidden layer returns $[(2, 1) - (0, 0)] = (2, 1)$ and updates its $\vec{x}_{\text{last}} \leftarrow (2, 1)$. It propagates through the corresponding weights and the resulting Σ_T of the output layer gets $[(2 \cdot 2 + (-3) \cdot 1)] = 1$ as input. Then Σ_T of the output layer returns $[(1) + (0)] = (1)$ and updates its $\vec{y_\Sigma} \leftarrow (1)$. After passing through the activation function, the final output is $\vec{y_1} = (1)$ which is the output at frame 1. At $t = 2$, the input $\vec{x} = (3.2, 1.1, 0)$ is first rounded to $(3, 1, 0)$. Then Δ_T of the input layer returns $[(3, 1, 0) - (1, 0, 0)] = (2, 1, 0)$ and updates its $\vec{x}_{\text{last}} \leftarrow (3, 1, 0)$. It propagates through the corresponding weights, and as a result

Σ_T of the hidden layer gets $[(2 \cdot 2 + 5 \cdot 1 + (-3) \cdot 0), (1 \cdot 2 + 2 \cdot 1 + 2 \cdot 0)] = (9, 4)$ as input. Then Σ_T of the hidden layer returns $[(9, 4) + (2, 1)] = (11, 5)$ and updates its $\vec{y_\Sigma} \leftarrow (11, 5)$. After passing through the activation function, we get $(h_1, h_2) = (11, 5)$. Then (h_1, h_2) after rounding remains (11, 5). Next, Δ_T of the hidden layer returns $[(11, 5) - (2, 1)] = (9, 4)$ and updates its $\vec{x}_{\text{last}} \leftarrow (11, 5)$. It propagates through the corresponding weights, and as a result Σ_T of the output layer gets $[(2 \cdot 9 + (-3) \cdot 4) = 6]$ as input. Then Σ_T of the output layer returns $[(6) + (1)] = (7)$ and updates its $\vec{y_\Sigma} \leftarrow (7)$. After passing through the activation function, we get $\vec{y_2} = (7)$ as the output at frame 2. At $t = 3$, the input $\vec{x} = (3.3, 0.9, 0.1)$ is first rounded to $(3, -1, 0)$. Then, Δ_T of the input layer returns $[(3, 1, 0) - (3, 1, 0)] = (0, 0, 0)$ and updates its $\vec{x}_{\text{last}} \leftarrow (3, 1, 0)$. It propagates through the corresponding weights, and as a result, Σ_T of the hidden layer gets $[(2 \cdot 0 + 5 \cdot 0 + (-3) \cdot 0), (1 \cdot 0 + 2 \cdot -2 + 2 \cdot 0)] = (0, 0)$ as input. Then, Σ_T of the hidden layer returns $[(0, 0) + (11, 5)] = (11, 5)$ and updates its $\vec{y_\Sigma} \leftarrow (11, 5)$. After passing through the activation function, we get $(h_1, h_2) = (11, 5)$. Then (h_1, h_2) after rounding remains $(11, 5)$. Next, Δ_T of the hidden layer returns $[(11, 5) - (11, 5)] = (0, 0)$ and updates its $\vec{x}_{\text{last}} \leftarrow (11, 5)$. It propagates through the corresponding weights, and as a result, Σ_T of the output layer gets $[(2 \cdot 0 + (-3) \cdot 0)] = 0$ as input. Then, Σ_T of the output layer returns $[(0) + (7)] = (7)$ and updates its $\vec{y_\Sigma} \leftarrow (7)$. After passing through the activation function, we get the final output $\vec{y_3} = (7)$. Here, we observe that when there is no significant change in the input features, the output of the SDNN remains unchanged, demonstrating its resistance to perturbations. This makes SDNNs popular. For $t = 4$ and 5, following the same process shown in tables 3.2, 3.3, 3.4, we obtain $y_4 = -1$, $y_5 = 8$. Thus, the complete output sequence over the temporal window is $\{y_t\}_{t=1}^{T} = \{1, 7, 7, -1, 8\}$. Now after the majority voting computation, we get $majority\_voting(\vec{y}) = 7$.

Extending this to an SDNN with multiple neurons in the output layer will give us multiple outputs—one from each neuron. To determine the final outcome, we pick

the neuron with the highest value at each time step, treating it as the most dominant response. Now, consider a classification task where our goal is to assign a label to an input sequence observed over a temporal window. Instead of making a decision at every instant, we could adopt a more robust approach—majority voting. Over time, as the SDNN processes the evolving sequence, each output step contributes a potential label, and by the end, we select the most frequently occurring label as the final decision. This ensures that even if the network's predictions vary slightly at individual time steps, the overall classification remains stable and reliable, capturing the essence of the temporal patterns in the input.

3.1 Verification Problem in SDNN

For a classification problem with m-classes, we can design an SDNN $\mathcal{N}$ with m output neurons. The SDNN predicts a class y for a given input sequence γ where y is the class corresponding to the neuron with the maximum vote in the output sequence η i.e. $\eta = \mathcal{N}(\gamma) \wedge y = majority\_voting(\eta)$. An SDNN $\mathcal{N}$ trained for classification is adversarially robust with respect to a given input γ_0 when small perturbations to γ_0 do not lead to a change in the predicted class $y_0 = majority\_voting(\eta_0)$ where $\eta_0 = \mathcal{N}(\gamma_0)$.

Definition 3.2 (**SDNN Verification Problem**)**.** For a SDNN $\mathcal{N} : \bar{x} \in \mathbb{R}^{n \times T} \rightarrow \bar{y} \in \mathbb{R}^{m \times T}$, an input property $P(\bar{x})$ and an output property $Q(\bar{y})$, the falsification problem is to find an input $\bar{x_0}$ with output $\bar{y_0} = \mathcal{N}(\bar{x_0})$, such that $\bar{x_0}$ satisfies P and $\bar{y_0}$ satisfies Q where $P(\bar{x})$ characterizes the inputs we are checking and $Q(\bar{y})$ characterizes the undesired behavior (i.e. violation of the requirement) for those inputs.

The outcome of the verification problem can be interpreted as follows:

- If no such input $\bar{x}_0$ is found (i.e., the search is unsuccessful), the result is **UNSAT**, indicating that the network satisfies the desired property and **the property holds**.
- If such an input is found, the result is **SAT**, indicating that the output exhibits unsafe behavior, and the input $\bar{x}_0$ serves as a **counterexample**.

Note that the dimensions of $\bar{x}$ and $\bar{y}$ are as mentioned earlier. For the verification framework, we need to encode a formula $P(\gamma) \wedge \mathcal{N}(\gamma) = \eta \wedge \neg Q(\eta)$ where $P(\gamma)$ states that γ is a perturbation of a given γ_0 with label y_0, and $Q(\eta)$ states that $majority\_voting(\eta) = y_0$. We now define our notion of adversarial robustness in SDNN.

3.2 Adversarial Robustness in SDNN

In the context of SDNN, adversarial robustness poses unique challenges due to the time-dependent nature of the inputs and the internal state of the network. Traditional adversarial perturbations, designed for static inputs, fail to capture the temporal dynamics present in temporal sequences. Hence, we introduce a formal notion of perturbation tailored specifically for SDNNs that respects both the temporal and spatial constraints of the input.

Definition 3.3 (Δ**-Perturbation**)**.** For given non-negative integers Δ, δ, ϵ, the Δ-Perturbation of a temporal input sequence γ_0 with temporal window T refers to changing it by at most Δ frames within T, where each frame can change by at most δ-many values within the frame, where each changed value within a frame can change by at most $\pm\epsilon$.

This definition allows us to model adversarial perturbations in a manner that aligns with the operational principles of SDNNs. By bounding the number of altered frames

(Δ), the number of values modified within each frame (δ), and the magnitude of each change (ϵ), we ensure that the perturbation remains subtle yet potentially adversarial. This formalism serves as the foundation for analyzing and verifying robustness properties in SDNNs.

3.3 Interval Bound Derivation

In the following discussion, we present a mechanism to derive bounds on the values that any Σ unit can take.

Lemma 3.4. *For an SDNN, consider consecutive Δ and Σ neurons at layers i and $j = i + 1$ respectively. Let x_i^t, Δ_i^t be the input and output of the Δ-neuron, and Σ_j^t be the output of the Σ-neuron at time t. Assume $\forall t \geq 1$, x_i^t is bounded as $\alpha \leq x_i^t \leq \beta$. Then, Σ_j^t is bounded as below:*

$$\sum_i w_{ij} \cdot \alpha + t \cdot b_j \leq \Sigma_j^t \leq \sum_i w_{ij} \cdot \beta + t \cdot b_j, \quad \forall t \geq 1$$

Proof: We know for an SDNN, at $t = 0$, all $x_i^0 = 0$, $\Sigma_j^0 = 0$, and for all timesteps $t \geq 1$ the following holds.

$$\Delta_i^t = x_i^t - x_i^{t-1} \tag{3.1}$$

$$\Sigma_j^t = \sum_i w_{ij} \cdot \Delta_i^t + b_j + \Sigma_j^{t-1} \tag{3.2}$$

where w_{ij} is the weight between node i and node j, and b_j is the bias corresponding to node j.

Base Case: at $t = 1$, for any $\alpha \leq x_i^1 \leq \beta$, we get $\alpha \leq \Delta_i^1 \leq \beta$ using eqn. 3.1. Then, eqn. 3.2 gives

$\sum_i w_{ij} \cdot \alpha + b_j \leq \Sigma_j^1 \leq \sum_i w_{ij} \cdot \beta + b_j$.

Induction Hypothesis: Let the statement be true for $t = k$ for $k \geq 2$, i.e.,

$$\sum_i w_{ij} \cdot \alpha + k \cdot b_j \leq \Sigma_j^k \leq \sum_i w_{ij} \cdot \beta + k \cdot b_j.$$

Induction Step: We need to show that the statement is true for $t = k + 1$ i.e.

$$\sum_i w_{ij} \cdot \alpha + (k+1) \cdot b_j \leq \Sigma_j^{k+1} \leq \sum_i w_{ij} \cdot \beta + (k+1) \cdot b_j.$$

At $t = k + 1$, by the assumption of the lemma:

$$\alpha \leq x_i^{k+1} \leq \beta$$

Then, by eqn. 3.1, $(\alpha - \beta) \leq \Delta_i^{k+1} \leq (\beta - \alpha)$.

By eqn. 3.2,

$$\Sigma_j^{k+1} = \sum_i w_{ij} \cdot \Delta_i^{k+1} + b_j + \Sigma_j^k$$

So, at $t = k + 1$,

$$\sum_i w_{ij} \cdot (\alpha - \beta) + b_j + \Sigma_j^k \leq \sum_i w_{ij} \cdot \Delta_i^{k+1} + b_j + \Sigma_j^k \leq \sum_i w_{ij} \cdot (\beta - \alpha) + b_j + \Sigma_j^k$$

$$\text{Or}, \sum_i w_{ij} \cdot (\alpha - \beta) + b_j + \sum_i w_{ij} \cdot \beta + k \cdot b_j \leq \sum_i w_{ij} \cdot \Delta_i^{k+1} + b_j + \Sigma_j^k \leq \sum_i w_{ij} \cdot (\beta - \alpha) + b_j + \sum_i w_{ij} \cdot \alpha + k \cdot b_j$$

$$\text{Or, } \sum_i w_{ij} \cdot (\alpha - \beta + \beta) + (k+1)b_j \leq \sum_i w_{ij} \cdot \Delta_i^{k+1} + b_j + \Sigma_j^k \leq \sum_i w_{ij} \cdot (\beta - \alpha + \alpha) + (k+1)b_j$$

$$\text{Or, } \sum_i w_{ij} \cdot \alpha + (k+1)b_j \leq \Sigma_j^{k+1} \leq \sum_i w_{ij} \cdot \beta + (k+1)b_j$$

Hence, by the principle of mathematical induction, the lemma is true for all $t \geq 1$. These bounds provide a structured framework for reasoning about adversarial perturbations in SDNNs. In Chapter 4, we leverage these bounds to derive the symbolic modeling of SDNNs for the purpose of verifying their adversarial robustness properties.

Chapter 4

Symbolic Modeling and Verification for SDNN

Symbolic modeling of SDNNs is one of our core contributions to address the adversarial robustness problem. Our framework takes two inputs: an SDNN and a property that needs verification. With these in hand, it sets out on one of two possible paths. On one path, the framework rigorously examines the SDNN and declares that the property holds true for all possible inputs. This assurance extends to inputs without restrictions as well as those constrained by specific input conditions defined in the property. On the other path, the framework uncovers a counterexample—an input where the property fails. This counterexample serves as a critical piece of evidence, showing exactly where the SDNN falls short. Thus, the framework acts as a certifier, either proving the reliability of the SDNN or revealing a vulnerability that demands attention. Through the SDNN verification problem, one can check for various system properties, such as adversarial robustness.

4.1 SMT Encoding of SDNN

The SMT encoding of a given SDNN $\mathcal{N}$ involves encoding the execution semantics of the internal and output neurons. Towards this, we first introduce the real variables used in our SMT encoding. For all input neurons N_i, let $x_{i,t}$ be the real variable denoting the value of the input corresponding to N_i at time t. For all other neurons N_i, let the variable $x_{i,t}$ denote the real output of N_i at time t. We present a modeling formalism for a given SDNN using SMT.

Constraints

For a given SDNN, let $w_{i,j} \in \mathbb{R}$ represent the weight of the synapse between the adjacent neurons (N_i, N_j). For all neurons $N_i \in N$, we define

$$\text{inSynapse}(N_i) = \{(N_j, N_i) \mid (N_j, N_i) \in \Psi\},$$

where $\text{inSynapse}(N_i)$ represents the incoming edges of N_i in the SDNN graph, Ψ denotes the edge set, and N denotes the vertex set of the SDNN graph. Furthermore, we introduce the following notation for each neuron N_i at time t:

- $r_{i,t}$: rounded real outputs for neuron N_i.
- $d_{i,t}$: output of the Δ module for neuron N_i.
- $y_{i,t}$: value of the integral of the Σ module for neuron N_i.

ξ_0***: Initialization :*** The input, output, quantized value, integral for each neuron are initialized to 0 at timestep 0.

$$\xi_0(i, 0) \triangleq (x_{i,0} = 0 \wedge r_{i,0} = 0 \wedge y_{i,0} = 0)$$

ξ_1: ***Quantization :*** For each neuron, the quantized value of the quantity stored in that neuron is calculated at timestep t.

$$\xi_1(i,t) \triangleq (x_{i,t} - [x_{i,t}] \geq 0.5 \implies r_{i,t} = [x_{i,t}] + 1) \wedge (x_{i,t} - [x_{i,t}] < 0.5 \implies r_{i,t} = [x_{i,t}])$$

$\xi_2 - \xi_4$: ***Encoding the*** $\Sigma - \Delta$ ***semantics*****:** The output of the Δ module for each neuron is calculated at timestep t.

$$\xi_2(i,t) \triangleq (d_{i,t} = r_{i,t} - r_{i,t-1})$$

During each timestep, when neuron N_i receives an input from the Δ module of neuron N_j, the product of the synaptic weight $w_{j,i}$ (connecting N_j to N_i) and the input is used to update the integral value of the Σ module of neuron N_i.

$$\xi_3(i,t) \triangleq \left(y_{i,t} = \sum_{j \in inSynapse(N_i)} (d_{j,t} \cdot w_{j,i}) + b_i + y_{i,t-1} \right)$$

We consider the activation function as the *ReLU* activation in our model. So, the output of the activation function for each neuron is calculated at timestep t.

$$\xi_4(i,t) \triangleq (y_{i,t} > 0 \implies o_{i,t} = y_{i,t}) \wedge (y_{i,t} \leq 0 \implies o_{i,t} = 0)$$

We use the constraints $\xi_0, \xi_1, \xi_2, \xi_3, \xi_4$ to model the SDNN verification problem discussed in Chapter 3.

4.2 Verifying Adversarial Robustness of SDNNs

For given non-negative integers Δ, δ, ϵ, the Δ-Perturbation of a temporal input sequence γ_0 with temporal window T refers to changing it by at most Δ frames

within T, where each frame can change by at most δ-many values within the frame, where each changed value within a frame can change by at most $\pm\epsilon$. If γ is obtained from the input γ_0 of temporal window of size T, then after Δ-perturbation, we have

Frame Perturbation Constraint:

$$\sum_{i=1}^{T} I_{(\gamma[i]\neq\gamma_0[i])} \leq \Delta \tag{4.1}$$

This ensures that at most Δ frames in the sequence γ_0 are modified. The indicator function $I_{(\gamma[i]\neq\gamma_0[i])}$ returns 1 if the i-th frame $\gamma[i]$ is different from $\gamma_0[i]$, and 0 otherwise. The summation guarantees that the number of perturbed frames does not exceed Δ.

Value Perturbation Constraint:

$$\bigwedge_{i=1}^{T} \left[I_{(\gamma[i]\neq\gamma_0[i])} \cdot \sum_{j=1}^{n} I_{(\gamma[i][j]\neq\gamma_0[i][j])} \leq \delta \right] \tag{4.2}$$

This condition applies to each frame in the temporal window of size T ($i = 1$ to T). The term n represents the total number of values in a frame, corresponding to the n inputs of the SDNN. The term $\sum_{j=1}^{n} I_{(\gamma[i][j]\neq\gamma_0[i][j])}$ returns 1 if the j-th value of the i-th frame $\gamma[i][j]$ is different from $\gamma_0[i][j]$, and 0 otherwise. It counts how many values have changed in the i-th frame. The inequality ensures that at most δ values are modified in each perturbed frame. The term $I_{(\gamma[i]\neq\gamma_0[i])}$ ensures that this constraint applies **only** to frames that were already identified as changed in the first condition. This applies more to temporal image datasets as shown later.

Value Intensity Constraint:

$$\bigwedge_{i=1}^{T} \bigwedge_{j=1}^{n} \|\gamma[i][j] - \gamma_0[i][j]\| \leq \epsilon \tag{4.3}$$

This ensures when a value in a frame is perturbed, its change is bounded. The

absolute difference between the jth value of the ith perturbed frame i.e. $\gamma[i][j]$ and the jth value of the ith original frame i.e. $\gamma_0[i][j]$ is at most ϵ. This prevents extreme modifications, ensuring that the extent of perturbation is bounded. The standard perturbation definitions in robustness analysis often focus on per-instance perturbations, these apply a global ϵ-bounded perturbation to all input dimensions. Our Δ-Perturbation framework introduces a new constraint structure, as described in the following:

Temporal constraints: We impose a **frame-level** restriction on perturbation count using Δ.

Frame group restrictions: Instead of unrestricted per-frame perturbations, we control the number of values changed in a frame using the upper bound of δ.

Limited per-value changes: We ensure that perturbed values do not exceed an intensity variation of ϵ.

For most real-life applications, the sequences generated by sensors are more susceptible to errors where there are missing intended values of pixels in frames or more values of pixels in frames than expected. So, we prefer to use the notion of Δ-Perturbation for adversarial robustness. Hence, the corresponding SMT encoding of the input property $P(\gamma)$ is denoted as F_P:

$$
\begin{aligned}
F_P \triangleq & \left[\sum_{i=1}^{n} I_{(\gamma[i]=\gamma_0[i])} \leq \Delta\right] \wedge \\
& \bigwedge_{i=1}^{n} \left[I_{(\gamma[i]=\gamma_0[i])} \cdot \sum_{j=1}^{|N_{inp}|} I_{(\gamma[i][j]=\gamma_0[i][j])} \leq \delta\right] \wedge \\
& \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{|N_{inp}|} \|\gamma[i][j] - \gamma_0[i][j]\| \leq \epsilon
\end{aligned} \tag{4.4}
$$

In order to formalize $Q(\eta), \forall t \in [1, \ldots, T]$, we encode $y_t = arg\_max(y_{1,t}, y_{2,t}, \ldots, y_{m,t})$ corresponding to the t-th column of η as:

$$F_{arg\_max}(\eta[t], y_t) \triangleq (y_t = j) \wedge (y_{j,t} \geq y_{i,t})\ \forall i, j \in \{1, \ldots, m\}$$

and the function $y = majority\_voting(y_1, y_2, \ldots, y_T)$ as:

$$F_{mv}(y_1, y2, \ldots, y_T, y_k) \triangleq \bigwedge_{t=1}^{T} (freq(y_k) \geq freq(y_t))$$

where $freq(y_k)$ denotes the frequency of the class y_k in the output sequence η. Recall in the notation used before, $majority\_voting(\eta) = y_0$. Therefore, the encoding of $Q(\eta)$ can be described as:

$$F_Q(\eta, y_0) \triangleq \bigwedge_{t=1}^{T} \big(F_{arg\_max}(\eta[t], y_t)\big) \wedge F_{mv}(y_1, y2, \ldots, y_T, y_0) \tag{4.5}$$

While the SMT formulation above does not assume any bounds except the domain bounds on the inputs, the derivation of interval bounds discussed in Chapter 3 gives an interesting result that helps us expedite the SMT even further by narrowing down the search it has to carry out while looking for a solution. This is another key contribution from us to SDNN literature, in addition to the symbolic modeling above. This scales up our SMT model even further as we show through our experiments in the following section.

We add the SMT encoding of the interval bounds of the Σ-unit in SDNNs for each node. If the bias is zero for any layer, then the corresponding bound on the Σ unit becomes constant.

ξ_5***: Bound Constraints :*** For each neuron, the integral value of its Σ-unit at timestep t can be encoded as:

$$\xi_5(i,t) \triangleq \left(y_{i,t} \geq \sum_{j \in inSynapse(N_i)} w_{j,i} \cdot \alpha + t \cdot b_j \right) \bigwedge \left(y_{i,t} \leq \sum_{j \in inSynapse(N_i)} w_{j,i} \cdot \beta + t \cdot b_j \right)$$

Collecting the formulas $\xi_0, \xi_1, \xi_2, \xi_3, \xi_4$ and ξ_5, we define the SMT encoding $F_{\mathcal{N}}$ of a given SDNN as below:

$$F_{\mathcal{N}} \triangleq \left(\bigwedge_{i \in n} \xi_0(i,0) \right) \wedge \left(\bigwedge_{t \in [1,T]} \bigwedge_{i \in n} \xi_1(i,t) \wedge \xi_2(i,t) \wedge \xi_3(i,t) \wedge \xi_4(i,t) \wedge \xi_5(i,t) \right) \quad (4.6)$$

To verify the adversarial robustness property of an SDNN, we send the verification query $(F_{\mathcal{P}} \wedge F_{\mathcal{N}} \wedge \neg F_{\mathcal{Q}})$ to our verification framework. If no input γ satisfies this formula, the result is **UNSAT**, indicating that the SDNN upholds the desired property under the given values of Δ, δ, and ϵ; in other words, the property holds. Conversely, if a satisfying input is found, the result is **SAT**, signifying a violation of the property. In this case, the corresponding input γ constitutes a **counterexample** that demonstrates unsafe behavior of the network.

Chapter 5

Implementation and Results

In this chapter, we present the results of our experiments on the temporal version of the MNIST dataset under 3 distinct experimental setups based on the parameters defined in Definition 3.3. Figure 5.1 illustrates an overall workflow of our proposed framework. Given an SDNN to check for adversarial robustness, a given temporal sequence γ_0 defined over a temporal window T, and a triplet of values Δ, δ, ϵ we run our adversarial example search by creating all possible Δ, δ, ϵ perturbations around γ_0 and check for robustness in each case using our SMT formulation. The framework outputs *Robust* if the set of constraints is *Unsatisfiable*, indicating that no such perturbation exists around the given γ_0 for which the SDNN outputs a different result. If the set of constraints is found *satisfiable*, it returns the counterexample giving us a valuation of Δ, δ, ϵ that can help us create a perturbed temporal sequence honoring the Δ, δ, ϵ constraint around γ_0 for which the given SDNN is non-robust, i.e. it outputs a different value. A third outcome is possible as well. If no counterexample is found within the specified time limit, the solver returns a timeout and the result of our analysis is inconclusive. Evidently, more the value of the time limit, more are the chances for deducing a conclusive result.

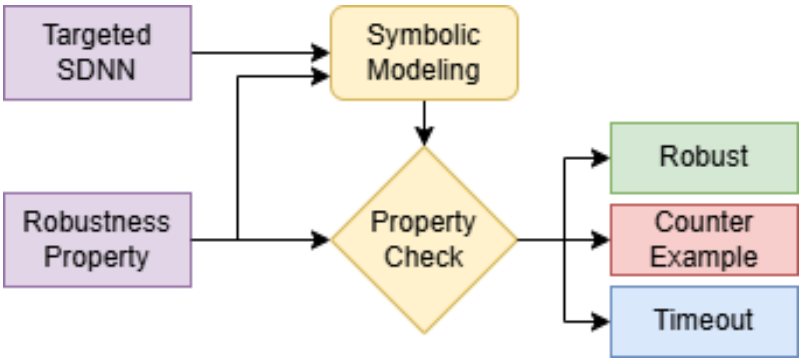

FIGURE 5.1: Our Proposed Framework

We now describe the implementation details followed by the experimental results. We used the MNIST dataset [32] to create temporal sequences. MNIST is a well-established benchmark dataset with simple yet diverse digit patterns, making it ideal for studying sequential transformations. Its gray-scale images allow for easy computation of pixel-wise similarities, enabling the creation of meaningful temporal sequences based on L1 distance [12].

5.1 Temporal MNIST

Temporal versions of the MNIST dataset [32] can be designed to facilitate testing and benchmarking models that require sequential data, such as SDNNs or SNNs. In [12], a variant of temporal MNIST is presented to test the SDNN. Here, we present a variant that is more suited to our definition of robustness. The dataset emphasizes spatio-temporal relationships by creating sequences of images based on their pixel-wise similarity, measured using the minimum L1 distance (sum of absolute pixel differences between images a, b, i.e. $|a - b|_{L_1} = \sum_i |a[i] - b[i]|$).

Creation of Temporal MNIST

The original MNIST dataset is partitioned into m *buckets*, where each bucket contains images corresponding to a specific digit (0–9) based on their labels. We create a

buffer of size N to generate the temporal sequences. To initialize the buffer, we perform sampling without replacement, selecting $\frac{N}{m}$ unique images for each class from the corresponding bucket. This ensures that no duplicate images are included within a bucket, preserving diversity while maintaining a balanced distribution across the classes. The generic algorithm behind our approach of temporal sequence generation is depicted in Algorithm 1.

Algorithm 1: Temporal Sequence Generation

Input: Dataset D containing m classes, Size of buffer N, Size of Temporal Window T

Output: Temporal *sequence* with its *label*

1: $sequence \leftarrow \{\}$
2: Partition dataset D into m buckets: $b_0, b_1, \ldots, b_{m-1}$ based on labels.
3: Initialize buffer $B \leftarrow \{B_0, B_1, \ldots, B_{m-1}\}$ where $B_i \leftarrow$ Randomly select $\frac{N}{m}$ unique images from b_i, $i \in \{0, \ldots, m-1\}$.
4: Randomly select an image i with *label* p from B.
5: **for** $t \leftarrow 1$ to T **do**
6: $\quad sequence \leftarrow sequence \cup i$
7: $\quad$ Replace i with j in B such that $i, j \in b_p$ and j is chosen randomly.
8: $\quad i \leftarrow \arg\min_{k \in B} |i - k|_{L_1}$
9: **end for**
10: label $\leftarrow \forall i \in sequence$, majority_voting(label($i$))
11: **return** *sequence*, *label*

In our implementation, with values $m = 10$, $N = 1000$, and $T = 16$, the generated Temporal MNIST dataset consists of sequences with a temporal window of size 16. Each frame in a sequence refers to a single grayscale image sampled from the original MNIST dataset. Note that, line 8 in Algorithm 1 selects the image k from the buffer B that has the minimum L_1 distance from the current image i. This operator, $\arg\min$, returns the argument (image k) that minimizes the distance. It ensures that the new image i selected for the next time step is the most visually similar (i.e., has the least pixel-wise difference) to the current image, thereby preserving temporal smoothness and enabling gradual transitions between frames in the generated sequence. Figure 5.2 shows a sample temporal image sequence of

length 16 with label 9 (outcome of majority voting), providing a visual representation of the dataset and aiding in the understanding of Algorithm 1.

FIGURE 5.2: Sample temporal sequence with label 9 from the MNIST dataset

5.2 Results on Adversarial Robustness

We utilize the SMT solver Z3 [30] (specifically, Z3 for Python Z3Py) to implement SMT encoding of SDNNs for verification. The verification framework requires three inputs: a pre-trained SDNN, an input sequence, and a triplet $(\Delta, \delta, \epsilon)$. In this dissertation, we report our evaluations on the Temporal MNIST dataset with temporal window sizes of 4, 8, 16, and 32. For our experiments, we employ SDNNs pre-trained on the Temporal MNIST dataset with the aforementioned temporal window sizes. The classification accuracies of these models are 92.32%, 92.71%, 93.85%, and 94.77%, respectively. If a counterexample is identified within the predefined timeout period, the framework reports the corresponding triplet $(\Delta, \delta, \epsilon)$ for which a counterexample to robustness is found.

Consider $F_{\mathcal{N}}$ in Equation 4.6. We use the term *vanilla SMT* to refer to $F_{\mathcal{N}}$ without the interval bound constraint ξ_5. As mentioned earlier, the version with the interval bound constraint is more efficient since the Σ units are bounded. For each of our experiments, we run both vanilla SMT and our SMT with Interval Bounds on a machine with 128 GB RAM and Ubuntu 22.04.

In our experiments, we set the timeout period to 1 hour. Each adversarial robustness experiment is conducted on 10 distinct sequences from the Temporal MNIST dataset, where the robustness of each such sequence is examined. The average time

required to reach a result over all 10 sequences is reported for each experiment, only considering the cases where a conclusive result (no timeout) is reached. We perform 3 distinct experimental setups based on the parameters defined in Definition 3.3. During our code implementation, all the fractional values corresponding to frame drops and frame perturbations are rounded in the following experiments.

5.2.1 Complete Frame Drop

In this set of experiments, one or more frames within the temporal window are removed entirely, and all values in each of the removed frames are set to zero. In this experiment, we fix the number of frames dropped (Δ) taking the other two parameters δ and ϵ as 100% in our model, to verify the robustness property. We then assess the robustness of the SDNN under these perturbation constraints. The results of this experiment are presented in Table 5.1, where the number of complete frame drops is expressed as a percentage.

Table 5.1 demonstrates that, for a fixed Δ, an increase in the temporal window size results in a decrease in the number of adversarial examples identified in SDNNs. Additionally, the SMT with Interval Bounds Framework outperforms the vanilla SMT in terms of average computation time, particularly for larger temporal sequences. The average time is computed as the mean of the times required to determine robustness or non-robustness across all 10 distinct sequences. The symbol "-" denotes instances where the result for each distinct sequence is inconclusive, leading to a timeout. Additionally, Table 5.1 shows that the SMT with the Interval Bounds Framework identifies more counterexamples within the timeout period when the sequence length is large. Furthermore, for small temporal window sizes, it detects counterexamples faster than the vanilla SMT framework.

Size of Temporal Window	Complete Frame Drop (%)	Adversarial Robustness with avg. time			
		Robust	Non-Robust	Avg. time (s)	Timeout
4	10	10, 10	0, 0	21.36, 21.17 ↑ 1.0x	0, 0
	20	4, 4	6, 6	84.31, 24.08 ↑ 3.5x	0, 0
	30	4, 4	6, 6	82.91, 24.33 ↑ 3.4x	0, 0
	40	3, 3	7, 7	82.97, 23.71 ↑ 3.5x	0, 0
8	10	10, 10	0, 0	85.91, 85.11 ↑ 1.0x	0, 0
	20	10, 10	0, 0	166.16, 141.68 ↑ 1.2x	0, 0
	30	10, 10	0, 0	241.82, 163.52 ↑ 1.5x	0, 0
	40	6, 6	4, 4	342.57, 264.79 ↑ 1.3x	0, 0
16	10	10, 10	0, 0	314.46, 277.34 ↑ 1.1x	0, 0
	20	10, 10	0, 0	328.90, 277.68 ↑ 1.2x	0, 0
	30	10, 10	0, 0	426.95, 339.04 ↑ 1.3x	0, 0
	40	6, 6	4, 4	580.47, 341.83 ↑ 1.7x	0, 0
32	10	10, 10	0, 0	219.24, 149.68 ↑ 1.5x	0, 0
	20	10, 10	0, 0	635.70, 541.83 ↑ 1.2x	0, 0
	30	0, 0	1, 2	3271.0, 2660.0 ↑ 1.2x	9, 8
	40	0, 0	0, 2	-, 3096.5	10, 8

TABLE 5.1: Performance on Temporal MNIST using vanilla SMT and SMT with Interval Bounds with different perturbation levels for various temporal window sizes. In each case, the average time has been calculated in seconds and the speedup of our model w.r.t. vanilla SMT is reported.

5.2.2 Partial Frame Drop with Complete Frame Perturbation

In this experiment, a subset of values within the chosen frames is set to zero instead of removing the entire frame, evaluating the network's response to partial occlusion. In this case, we fix 2 parameters- the maximum number of frames to be partially dropped Δ and the number of inputs perturbed in those frames (δ) taking ϵ as 100% in our model, to verify the robustness property. The results of this experiment is shown in Table 5.2, where the number of partial frame drops and the number of complete frame perturbations are expressed as percentages.

Table 5.2 demonstrates that, for the number of perturbed frames less than or equal to Δ and a fixed δ, an increase in the temporal window size results in a decrease in the number of adversarial examples identified in SDNNs. In this case as well, the average

Size of Temporal Window	Partial Frame Drop (%)	Complete Frame Perturbation (%)	Adversarial Robustness with avg. time			
			Robust	Non-Robust	Avg. time.	Timeout
4	10	10	10, 10	0, 0	121.51, 82.66 ↑ 1.5x	0, 0
	10	15	10, 10	0, 0	121.65, 82.69 ↑ 1.5x	0, 0
	10	20	10, 10	0, 0	151.27, 92.28 ↑ 1.6x	0, 0
	10	25	10, 10	0, 0	222.04, 183.36 ↑ 1.2x	0, 0
	20	10	10, 10	0, 0	224.47, 183.64 ↑ 1.2x	0, 0
	20	15	10, 10	0, 0	227.05, 194.97 ↑ 1.2x	0, 0
	20	20	10, 10	0, 0	282.10, 267.19 ↑ 1.0x	0, 0
	20	25	10, 10	0, 0	326.56, 282.09 ↑ 1.2x	0, 0
	30	10	10, 10	0, 0	341.90, 293.97 ↑ 1.2x	0, 0
	30	15	10, 10	0, 0	447.68, 372.91 ↑ 1.2x	0, 0
	30	20	10, 10	0, 0	496.59, 443.84 ↑ 1.1x	0, 0
	30	25	10, 10	0, 0	636.67, 564.71 ↑ 1.1x	0, 0
	40	10	10, 10	0, 0	612.22, 423.83 ↑ 1.4x	0, 0
	40	15	10, 10	0, 0	1602.84, 964.96 ↑ 1.6x	0, 0
	40	20	9, 9	1, 1	2609.18, 1574.43 ↑ 1.6x	0, 0
	40	25	8, 8	0, 1	3428.20, 2953.46 ↑ 1.1x	2, 1
8	10	10	10, 10	0, 0	217.39, 162.92 ↑ 1.3x	0, 0
	10	15	10, 10	0, 0	284.31, 165.05 ↑ 1.7x	0, 0
	10	20	10, 10	0, 0	294.97, 174.41 ↑ 1.6x	0, 0
	10	25	10, 10	0, 0	362.91, 265.96 ↑ 1.4x	0, 0
	20	10	10, 10	0, 0	421.17, 266.53 ↑ 1.6x	0, 0
	20	15	10, 10	0, 0	480.47, 365.79 ↑ 1.3x	0, 0
	20	20	10, 10	0, 0	635.69, 469.63 ↑ 1.3x	0, 0
	20	25	10, 10	0, 0	728.89, 502.31 ↑ 1.4x	0, 0
	30	10	10, 10	0, 0	635.72, 466.73 ↑ 1.4x	0, 0
	30	15	10, 10	0, 0	893.96, 563.95 ↑ 1.6x	0, 0
	30	20	10, 10	0, 0	922.04, 684.06 ↑ 1.3x	0, 0
	30	25	10, 10	0, 0	1326.55, 866.2 ↑ 1.5x	0, 0
	40	10	10, 10	0, 0	780.46, 669.44 ↑ 1.2x	0, 0
	40	15	10, 10	0, 0	926.94, 768.21 ↑ 1.2x	0, 0
	40	20	9, 9	1, 1	2602.18, 1890.54 ↑ 1.4x	0, 0
	40	25	6, 7	2, 2	3486.98, 3295.49 ↑ 1.1x	2, 1

TABLE 5.2: Performance on Temporal MNIST using vanilla SMT and SMT with Interval Bounds with different perturbation levels for various temporal window sizes. In each case, the average time has been calculated in seconds and the speedup of our model w.r.t. vanilla SMT is reported.

time is computed in the same manner as in the previous experiment. The symbol "-" denotes instances where the result for each distinct sequence is inconclusive, leading to a timeout. We observe that this experiment requires more computational time compared to the previous one, indicating a higher complexity, which is expected.

5.2.3 Partial Frame Drop with Partial Frame Perturbation and Partial Pixel Perturbation

In this experiment, selected pixels within the dropped frames are perturbed within a predefined range ϵ, further testing the network's robustness to fine-grained modifications. We fix 3 parameters-the maximum number of partially dropped frames (Δ), the maximum number of inputs perturbed in those frames (δ) and the range of the perturbations (ϵ) to the model to test the network's robustness to more fine-grained perturbations. The results of this experiment is shown in Table 5.3 (for temporal window of size 4) and Table 5.4 (for window size 8).

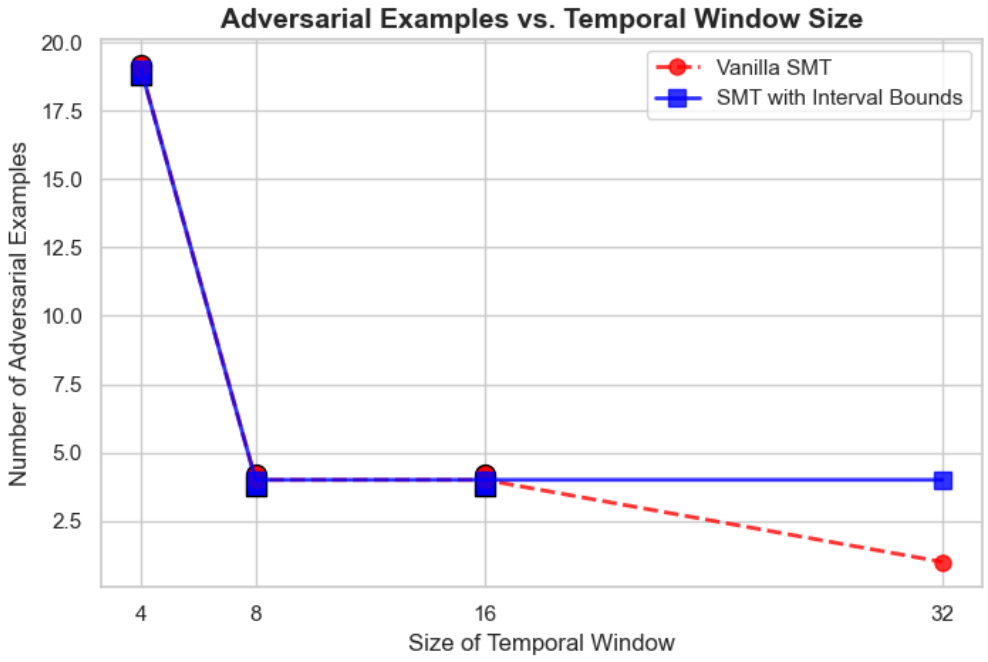

FIGURE 5.3: Result on Temporal MNIST for Experiment 1

Table 5.3 shows that when the temporal window size is small, the SMT with Interval Bounds detects counterexamples faster than the vanilla SMT framework. In this case, the average time is computed in the same manner as in the previous experiments. We also observe that this experiment requires more computational time compared

Partial Frame Drop (%)	Partial Frame Perturbation (%)	Partial Pixel Perturbation (ϵ)	Adversarial Robustness with avg. time			
			Robust	Non-Robust	Avg. time	Timeout
10	10	10	10, 10	0, 0	221.35, 221.17 ↑ 1.0x	0, 0
		20	10, 10	0, 0	221.49, 220.95 ↑ 1.0x	0, 0
		30	10, 10	0, 0	222.02, 221.16 ↑ 1.0x	0, 0
		40	10, 10	0, 0	221.64, 220.84 ↑ 1.0x	0, 0
	15	10	10, 10	0, 0	221.13, 220.89 ↑ 1.0x	0, 0
		20	10, 10	0, 0	221.41, 220.87 ↑ 1.0x	0, 0
		30	10, 10	0, 0	221.56, 220.93 ↑ 1.0x	0, 0
		40	10, 10	0, 0	221.42, 221.03 ↑ 1.0x	0, 0
	20	10	10, 10	0, 0	222.04, 221.50 ↑ 1.0x	0, 0
		20	10, 10	0, 0	221.51, 221.16 ↑ 1.0x	0, 0
		30	10, 10	0, 0	221.37, 221.06 ↑ 1.0x	0, 0
		40	10, 10	0, 0	221.64, 220.84 ↑ 1.0x	0, 0
	25	10	10, 10	0, 0	221.82, 220.98 ↑ 1.0x	0, 0
		20	10, 10	0, 0	221.49, 221.00 ↑ 1.0x	0, 0
		30	10, 10	0, 0	221.32, 220.85 ↑ 1.0x	0, 0
		40	10, 10	0, 0	221.05, 221.64 ↑ 1.0x	0, 0
20	10	10	10, 10	0, 0	525.27, 493.96 ↑ 1.1x	0, 0
		20	10, 10	0, 0	626.73, 583.69 ↑ 1.1x	0, 0
		30	10, 10	0, 0	728.74, 683.08 ↑ 1.1x	0, 0
		40	10, 10	0, 0	823.70, 782.77 ↑ 1.1x	0, 0
	15	10	10, 10	0, 0	768.21, 681.94 ↑ 1x	0, 0
		20	10, 10	0, 0	919.23, 783.63 ↑ 1.2x	0, 0
		30	10, 10	0, 0	1164.54, 882.75 ↑ 1.3x	0, 0
		40	10, 10	0, 0	1602.83, 983.74 ↑ 1.6x	0, 0
	20	10	10, 10	0, 0	982.10, 871.16 ↑ 1.1x	0, 0
		20	10, 10	0, 0	1580.48, 1021.51 ↑ 1.5x	0, 0
		30	10, 10	0, 0	2194.98, 1621.50 ↑ 1.3x	0, 0
		40	10, 10	0, 0	3196.50, 2620.85 ↑ 1.2x	0, 0
	25	10	10, 10	0, 0	2219.25, 1721.63 ↑ 1.3x	0, 0
		20	10, 10	0, 0	3071.15, 2420.94 ↑ 1.3x	0, 0
		30	5, 7	2, 2	3096.49, 2721.64 ↑ 1.1x	3, 1
		40	0, 0	0, 1	-, 3251.2	10, 9

TABLE 5.3: Performance on Temporal MNIST with temporal window of size 4 using vanilla SMT and SMT with Interval Bounds with different perturbation levels. The average time has been calculated in seconds. The speed up in the performance of our model w.r.t vanilla SMT is shown.

to the previous two experiments, indicating a higher complexity, which is expected. Additionally, Table 5.3 shows that the SMT with Interval Bounds identifies more counterexamples within the timeout period when the value of Δ is large. This highlights the greater efficiency of the SMT with Interval Bounds compared to the vanilla SMT framework. Table 5.4 also shows that when the temporal window size is small, the SMT with Interval Bounds Framework detects counterexamples faster than the vanilla SMT framework. However, for temporal window sizes of 16 and

32, both the vanilla SMT and the SMT with Interval Bounds timeout for all test sequences.

Partial Frame Drop (%)	Partial Frame Perturbation (%)	Partial Pixel Perturbation (δ)	Adversarial Robustness with avg. time			
			Robust	Non-Robust	Avg. time	Timeout
10	10	10	10, 10	0, 0	521.51, 463.49 ↑ 1.0x	0, 0
		20	10, 10	0, 0	621.49, 520.95 ↑ 1.2x	0, 0
		30	10, 10	0, 0	692.02, 571.16 ↑ 1.2x	0, 0
		40	10, 10	0, 0	721.64, 620.84 ↑ 1.2x	0, 0
	15	10	10, 10	0, 0	525.26, 471.15 ↑ 1.1x	0, 0
		20	10, 10	0, 0	680.46, 649.67 ↑ 1.0x	0, 0
		30	10, 10	0, 0	847.64, 782.75 ↑ 1.1x	0, 0
		40	10, 10	0, 0	1094.31, 983.73 ↑ 1.1x	0, 0
	20	10	10, 10	0, 0	982.08, 772.56 ↑ 1.3x	0, 0
		20	10, 10	0, 0	1121.41, 877.06 ↑ 1.3x	0, 0
		30	10, 10	0, 0	1267.53, 983.63 ↑ 1.3x	0, 0
		40	10, 10	0, 0	1571.14, 1219.25 ↑ 1.3x	0, 0
	25	10	10, 10	0, 0	921.48, 883.64 ↑ 1.0x	0, 0
		20	10, 10	0, 0	994.30, 921.50 ↑ 1.1x	0, 0
		30	10, 10	0, 0	1083.64, 1017.21 ↑ 1.1x	0, 0
		40	10, 10	0, 0	1702.82, 1560.01 ↑ 1.1x	0, 0
20	10	10	10, 10	0, 0	621.32, 593.95 ↑ 1.0x	0, 0
		20	10, 10	0, 0	720.85, 625.28 ↑ 1.2x	0, 0
		30	10, 10	0, 0	971.18, 823.69 ↑ 1.2x	0, 0
		40	10, 10	0, 0	1081.24, 1019.09 ↑ 1.1x	0, 0
	15	10	10, 10	0, 0	882.78, 721.63 ↑ 1.2x	0, 0
		20	10, 10	0, 0	945.70, 820.95 ↑ 1.2x	0, 0
		30	10, 10	0, 0	999.34, 920.25 ↑ 1.1x	0, 0
		40	10, 10	0, 0	1291.62, 1051.19 ↑ 1.2x	0, 0
	20	10	10, 10	0, 0	830.58, 768.20 ↑ 1.1x	0, 0
		20	10, 10	0, 0	923.49, 871.15 ↑ 1.1x	0, 0
		30	10, 10	0, 0	1016.29, 963.50 ↑ 1.1x	0, 0
		40	10, 10	0, 0	1820.69, 1420.95 ↑ 1.3x	0, 0
	25	10	10, 10	0, 0	934.20, 820.26 ↑ 1.1x	0, 0
		20	10, 10	0, 0	1156.29, 971.17 ↑ 1.2x	0, 0
		30	10, 10	0, 0	2023.68, 1521.50 ↑ 1.3x	0, 0
		40	8, 10	0, 0	3499.67, 2621.69 ↑ 1.3x	2, 0

TABLE 5.4: Performance on Temporal MNIST with temporal window of size 8 using vanilla SMT and SMT with Interval Bounds with different perturbation levels. The average time has been calculated in seconds. The speed up in the performance of our model w.r.t vanilla SMT is shown.

As we can see from the results, the SMT formulation with the interval bound constraints performs better than vanilla SMT. The speedup increases as we move to more complex perturbations and greater temporal window sizes. Another interesting observation coming out from our first set of experiments is shown in Figure 5.3. As the size of the temporal window increases, the number of adversarial examples found

in SDNNs decreases. This indicates that SDNNs become more robust to adversarial attacks when larger temporal contexts are considered, suggesting that temporal dependencies help to improve the stability and security of the network.

Chapter 6

PilotNet - An Advanced SDNN with Convolution Operation

In the previous chapter, we formally defined the verification problem for Sigma-Delta Neural Networks (SDNNs) by characterizing it as a constraint satisfaction problem (CSP). We discussed how properties over temporally varying input-output sequences can be used to express safety requirements, and how falsification attempts to identify unsafe counterexamples that violate these requirements.

Building on this foundation, we now introduce **PilotNet**, an advanced SDNN architecture specifically designed for processing high-dimensional visual inputs using convolutional layers. While basic SDNN models demonstrate temporal sparsity and efficient event-driven processing, they typically rely on fully connected layers and are limited in their ability to process spatially structured data like images. PilotNet addresses this limitation by incorporating convolution operations into the SDNN framework, enabling it to efficiently process spatio-temporal features over time.

In this chapter, we make the following contributions:

- We describe a **modified Delta module** with an adaptive thresholding mechanism that regulates the firing of spikes based on signal variation and ensures

better temporal precision.

- We explain the **Convolution Operation** used within the Sigma-Delta paradigm, detailing how it extends standard convolutional layers to operate on spike-coded inputs.
- We present the **PilotNet architecture** and discuss its significance as a benchmark SDNN for visual tasks, particularly in the context of autonomous driving and perception modules.
- We propose a symbolic bound computation strategy for PilotNet, enabling formal reasoning over its output in the presence of temporally evolving inputs. This allows verification of temporal properties and robustness guarantees.
- We perform the SMT-based symbolic encoding of the PilotNet architecture, translating its sigma-delta procedures and convolution operation into logical constraints suitable for formal verification.

The introduction of convolution operations in SDNNs not only enhances their expressiveness and applicability but also presents new challenges in verification. The increased complexity in spatio-temporal representations necessitates more refined abstractions and efficient formal analysis methods, which we explore in the subsequent chapters.

6.1 Modifications to Sigma-Delta Neural Networks

In many real-world applications, inputs and intermediate activations in neural networks are temporally correlated. Traditional ANNs, however, process each input independently due to their stateless design, which leads to considerable redundant computation—especially when dealing with time-series or streaming data.

Sigma Delta Neural Networks (SDNNs) mitigate this inefficiency by encoding only the changes in data over time. Instead of transmitting raw values, SDNNs communicate *temporal deltas*, exploiting the inherent redundancy between successive frames or time steps. This selective communication significantly reduces computational overhead and energy consumption.

The **delta module** transmits a graded spike $s[t]$ at time t, only if the deviation of the current value $x[t]$ from the previous reference value $x_{\text{ref}}[t-1]$ exceeds a predefined threshold η:

$$s[t] = (x[t] - x_{\text{ref}}[t-1]) \cdot H(|x[t] - x_{\text{ref}}[t-1]| - \eta) \tag{6.1}$$

$$x_{\text{ref}}[t] = x_{\text{ref}}[t-1] + s[t] \tag{6.2}$$

where $H(\cdot)$ is the Heaviside step function [33]. The thresholding mechanism enforces sparsity by ignoring insignificant variations.

The modification introduced in the following Temporal Difference procedure lies specifically in the calculation of the delta module, where thresholding and a residual accumulator are integrated into the computation. Unlike the SDNN framework described in Chapter 3, which may not explicitly include residual tracking, this formulation ensures that any sub-threshold variations are preserved and accumulated over time. This guarantees that even small changes eventually contribute to the output once they surpass the threshold η, thereby maintaining fidelity while enforcing sparsity more effectively.

6.2 Convolution Operation in PilotNet

Convolution is a fundamental operation in Convolutional Neural Networks (CNNs), commonly used to extract spatial hierarchies and features from input data such as

Procedure Temporal Difference(Δ_T)

Internal: $\vec{x}_{\text{last}} \in \mathbb{R}^d \leftarrow \vec{0}, \vec{x}_{\text{residual}} \in \mathbb{R}^d \leftarrow \vec{0}$
Input: $\vec{x} \in \mathbb{R}^d$, threshold (η)
Output: $\vec{y_\Delta} \in \mathbb{R}^d$
1: $delta \leftarrow \vec{x} - \vec{x}_{\text{last}} + \vec{x}_{\text{residual}}$
2: $\vec{y_\Delta} \leftarrow delta\ if\ |delta| > \eta, else\ 0$
3: $\vec{x}_{\text{residual}} \leftarrow delta - \vec{y_\Delta}$
4: $\vec{x}_{\text{last}} \leftarrow \vec{x}$
5: **return** $\vec{y_\Delta}$

images. The convolution process involves sliding a small matrix called a **kernel** (or filter) over the input tensor to produce a corresponding **feature map**.

Key Parameters of Convolution Operation

- **Kernel (or Filter):** A small-sized matrix (e.g., 3×3, 5×5) used to compute dot products with overlapping regions of the input. The kernel contains learnable weights that are optimized during training.

- **Stride (s):** The number of pixels by which the kernel is moved across the input. A stride of 1 means the kernel moves one pixel at a time. Larger strides downsample the output feature map. The output dimension (in 1D) is given by:

$$\text{Output size} = \left\lfloor \frac{\text{Input size} - \text{Kernel size}}{\text{Stride}} \right\rfloor + 1$$

- **Padding (p):** Padding adds extra pixels (typically zeros) around the border of the input to control the spatial size of the output. It helps to preserve the spatial resolution. Common types include:

  – *Valid padding:* No padding, resulting in a smaller output.

  – *Same padding:* Padding is added to keep the output size equal to the input size.

- **Dilation (d):** The spacing between the elements of the kernel. A dilation rate of 1 represents a standard convolution. Larger dilation increases the receptive field without increasing the kernel size. It is useful in dilated (or atrous) convolutions for capturing multi-scale context.

- **Input and Output Channels:** The input to a convolution layer has a certain number of channels (e.g., 3 for RGB images). The number of filters used determines the number of output channels or feature maps, where each filter learns a different feature representation.

- **Receptive Field:** The region in the input space that influences a specific value in the output feature map. Larger receptive fields enable the network to capture more contextual information.

In summary, there are two inputs to a convolutional operation:

1. A 3D volume (input image) of size ($n_{\text{in}} \times n_{\text{in}} \times$ channels)

2. A set of k filters (also called kernels or feature extractors), each of size ($f \times f \times$ channels), where f is typically 3 or 5.

The output of a convolutional operation is also a 3D volume (also called an output image or feature map) of size ($n_{\text{out}} \times n_{\text{out}} \times k$).

The relationship between n_{in} and n_{out} is as follows:

$$n_{\text{out}} = \left\lfloor \frac{n_{\text{in}} + 2p - f}{s} \right\rfloor + 1$$

where:

- n_{in}: number of input features

- n_{out}: number of output features

- f: convolution kernel size
- p: convolution padding size
- s: convolution stride size

Convolution operation can be visualized in Figure 6.1.

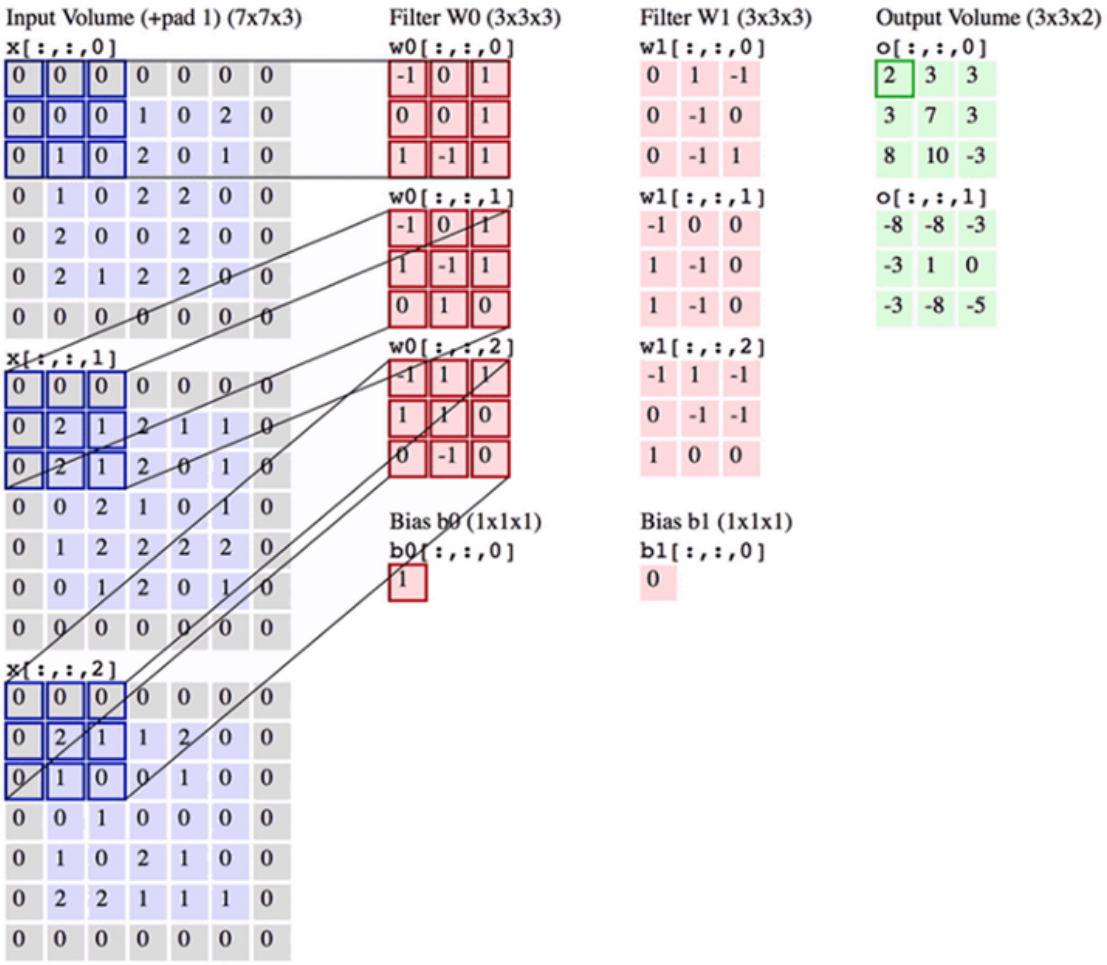

FIGURE 6.1: Convolution with 3 input channels, 2 output channels, 2 filters $f = 3$, stride $s = 2$, padding $p = 1$, dilation $d = 1$.

6.3 PilotNet

PilotNet is an end-to-end deep learning system developed for autonomous driving, designed to predict a vehicle's steering angle from dashboard-mounted RGB camera input [1, 13]. Driving scenarios typically involve high temporal continuity i.e. consecutive frames are visually similar with only small differences. This makes PilotNet an ideal candidate for sigma-delta sparsification. By integrating sigma-delta encapsulation into the network, we can minimize redundant processing of similar frames, thereby improving computational efficiency without degrading the prediction accuracy much. This work leverages the synergy between SDNNs and PilotNet to explore robust, sparse, and temporally efficient neural computation in safety-critical domains.

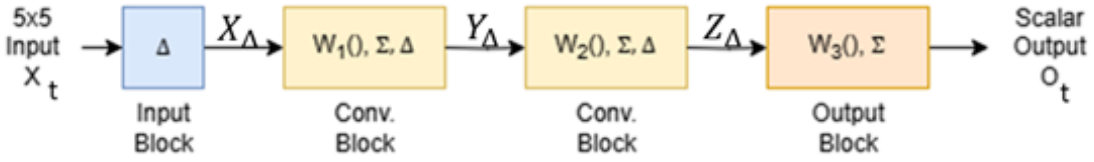

FIGURE 6.2: SDNN based PilotNet example

The following example explains the philosophy behind working of an SDNN based PilotNet architecture:

Example 6.1. *Consider a simple SDNN based PilotNet model with* 5×5 *input features, 1 input block, 2 convolution blocks, 1 output block, 1 output feature shown in Figure 6.2. The input block consists of a single* Δ *operation and outputs* X_Δ *of size* 5×5*. The convolution blocks includes the convolution operation followed by one set of* Σ *and* Δ *operations which output* Y_Δ, Z_Δ *of size* 5×5, 3×3 *respectively. In the output block, there is an element-wise dot product followed by a* Σ *operation that outputs* o_t*. The first convolution block performs the convolution operation with padding=1, stride=1 while, the second convolution block performs the convolution*

operation with padding=1, stride=2. Let for input $\vec{x} \in \mathbb{R}^{5\times5\times T}$*, that is, at each time step* $t \in [1, T]$*, the system receives an input vector* $\vec{x}_t \in \mathbb{R}^{5\times5}$ *consisting of the input features. The output at time step* t *is denoted as* $o_t \in \mathbb{R}$*.*

$$\mathrm{X}_1 = \begin{bmatrix} 0 & 0 & 1 & 0 & 2 \\ 1 & 0 & 2 & 0 & 1 \\ 1 & 0 & 2 & 2 & 0 \\ 2 & 0 & 0 & 2 & 0 \\ 2 & 1 & 2 & 2 & 0 \end{bmatrix} \quad X_2 = \begin{bmatrix} 0.2 & 0.3 & 0.9 & 0.7 & 1.8 \\ 1.2 & 1.2 & 0.8 & 0.1 & 1.0 \\ 0.0 & 0.5 & 1.3 & 1.9 & 0.0 \\ 1.0 & 0.0 & 0.1 & 2.0 & 0.0 \\ 2.1 & 0.9 & 2.0 & 2.0 & 0.1 \end{bmatrix} \quad X_3 = \begin{bmatrix} 0.2 & 0.5 & 1.3 & 0.4 & 1.9 \\ 1.4 & 1.2 & 0.8 & 0.1 & 1.1 \\ 0.3 & 0.5 & 1.3 & 1.9 & 0.0 \\ 1.0 & 0.0 & 0.1 & 2.1 & 0.0 \\ 2.2 & 0.9 & 2.0 & 2.5 & 0.1 \end{bmatrix}$$

FIGURE 6.3: Three input matrices: X_1, X_2, and X_3

Three consecutive inputs, which are given to the model shown in Figure 6.2, are shown in Figure 6.3. The threshold values for the Δ *operations are 0.5, 1.0, and 1.0 in the input block and two convolution blocks, respectively. Note that the threshold value is compared element-wise in the* Δ *operation. The weight matrices* W_1*,* W_2*, used in 2 convolution operations, and* W_3 *used in the output block are shown in Figure 6.4.*

$$W_1 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 2 \\ 1 & 0 & 2 \end{bmatrix} \quad W_2 = \begin{bmatrix} -1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & -1 & 1 \end{bmatrix} \quad W_3 = \begin{bmatrix} -1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & -1 & 1 \end{bmatrix}$$

FIGURE 6.4: Three weight matrices: W_1, W_2, and W_3

At $t = 1$*, the input* X_1 *is first processed by the input block which returns* X^1_Δ *with the exact values of* X_1 *as initially the value of* X_0 *and* $X_{residual}$ *are zeros and all the values of* $X_1 \geq 0.5$*. The first and second convolution block returns* Y^1_Δ *and* Z^1_Δ *respectively after doing the convolution,* Σ *and* Δ *operations shown in Figure 6.5. The output block returns* -2*.*

At $t = 2$*, the input* X_2 *is first processed by the input block which returns* X^2_Δ *followed by the first and second convolution block which return* Y^2_Δ *and* Z^2_Δ *respectively shown in Figure 6.7. The output block returns* 2.8*.*

$$\mathrm{X}^1_\Delta = \begin{bmatrix} 0 & 0 & 1 & 0 & 2 \\ 1 & 0 & 2 & 0 & 1 \\ 1 & 0 & 2 & 2 & 0 \\ 2 & 0 & 0 & 2 & 0 \\ 2 & 1 & 2 & 2 & 0 \end{bmatrix} \quad Y^1_\Delta = \begin{bmatrix} 0 & 5 & 0 & 8 & 0 \\ 0 & 9 & 2 & 8 & 2 \\ 0 & 9 & 6 & 3 & 4 \\ 1 & 8 & 9 & 2 & 4 \\ 2 & 6 & 7 & 2 & 2 \end{bmatrix} \quad Z^1_\Delta = \begin{bmatrix} 14 & 23 & 6 \\ 25 & 3 & -10 \\ 14 & -4 & -2 \end{bmatrix}$$

FIGURE 6.5: Intermediate matrices at $t = 1$: X_Δ, Y_Δ, and Z_Δ

$$\mathrm{X}^2_\Delta = \begin{bmatrix} 0 & 0 & 0 & 0.7 & 2 \\ 0 & 1.2 & -1.2 & 0 & 1 \\ -1.0 & 0.5 & -0.7 & 2 & 0 \\ -1.0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad Y^2_\Delta = \begin{bmatrix} 1.2 & -1.2 & 2.6 & -1.2 & 0 \\ 2.9 & -4.1 & 2.4 & -1.9 & 0 \\ 2.2 & -4.6 & 0 & 0 & 0 \\ 0 & -1.7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad Z^2_\Delta = \begin{bmatrix} -8.2 & -9.6 & -1.9 \\ -10.4 & 0 & 1.9 \\ -1.7 & 1.7 & 0 \end{bmatrix}$$

FIGURE 6.6: Intermediate matrices at $t = 2$: X_Δ, Y_Δ, and Z_Δ

At $t = 3$, the input X_3 is first processed by the input block which returns X^3_Δ, followed by the first and second convolution blocks which return Y^3_Δ and Z^3_Δ respectively shown in Figure 6.7. The output block returns 2.8. *Here, we observe that when there is no significant change in the input features, the output of the above SDNN based PilotNet model remains unchanged, demonstrating its resistance to perturbations. This makes SDNNs popular to be used in PilotNet.*

$$\mathrm{X}^3_\Delta = \begin{bmatrix} 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0 \end{bmatrix} \quad Y^3_\Delta = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \quad Z^3_\Delta = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

FIGURE 6.7: Intermediate matrices at $t = 3$: X_Δ, Y_Δ, and Z_Δ

$$\mathrm{X}_{residual} = \begin{bmatrix} 0.2 & 0.3 & -0.1 & 0 & -0.2 \\ 0.2 & 0 & 0 & 0.1 & 0 \\ 0 & 0 & 0 & -0.1 & 0 \\ 0 & 0 & 0.1 & 0 & 0 \\ 0.1 & -0.1 & 0 & 0 & 0.1 \end{bmatrix} \quad Y_{residual} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0.7 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & -0.7 & 0 \\ 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad Z_{residual} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0.7 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

FIGURE 6.8: Residual matrices at $t = 2$: $X_{residual}$, $Y_{residual}$, $Z_{residual}$

We can compute the residual matrices for each of the Δ operations used in the input and the convolution blocks. For example, at $t = 2$, after each block's Δ operation, the residual matrices are $X_{residual}$, $Y_{residual}$, $Z_{residual}$ shown in Figure 6.8.

Extending this setup to an SDNN based PilotNet model with multiple convolution and dense layers will give us a single output i.e. the steering angle of the vehicle at a particular time step.

6.4 Bound Propagation through PilotNet

In this section, we formally analyze how input perturbations affect the outputs of each module in the PilotNet architecture. The objective is to establish provable bounds on the network's activations under bounded uncertainty, which is a key step in formal verification. We begin with the Δ-module, which introduces nonlinearity and sparsity into the temporal encoding.

6.4.1 Bound Propagation through the Δ-Module

The Δ-module emits an output only when the change in the input signal exceeds a threshold. This introduces conditional behavior that can impact the sign and magnitude of the output under perturbations. Understanding how bounded perturbations affect the sign of Δ-outputs is critical in preserving decision robustness in downstream layers.

Assume that the values in the input sequence γ_0 are bounded such that

$$\alpha \leq x_i^t \leq \beta, \quad \forall x_i^t \in \gamma_0.$$

Let the threshold of the Δ-module be a nonnegative scalar $\mu \geq 0$. Then, in the absence of perturbation, the output Δ_j^t of the Δ-module is bounded as:

$$\left(\Delta_j^t = 0\right) \vee \left((\alpha - \beta + \mu) \leq \Delta_j^t \leq -\mu\right) \vee \left(\mu \leq \Delta_j^t \leq (\beta - \alpha + \mu)\right) \tag{6.3}$$

To reason about the robustness of the Δ-module under input perturbation, we now state the following lemma:

Lemma 6.1 (**Sign Invariance under Perturbations**). *Let the perturbation on the input be bounded such that $\epsilon < \mu$. Then, for any time step t and any index j, the output Δ_j^t of the Δ-module cannot change its sign due to the perturbation. In other words, a negative Δ_j^t cannot become positive, and vice versa.*

Proof Assume there exists an output Δ_j^t that is originally negative and becomes positive under a perturbation $\epsilon < \mu$.

Let $\Delta_j^t = -\mu$ (maximum negative value emitted by the Δ-module). For this to flip sign and become positive, the net increase due to perturbation must exceed 2μ. This is because both the current value $x[t]$ and the reference value $x_{\text{ref}}[t-1]$ could shift in opposite directions, leading to a total change of at most 2ϵ.

But since $\epsilon < \mu$, we have:

$$\Delta_j^t + 2\epsilon = -\mu + 2\epsilon < \mu.$$

Thus, the value remains strictly less than the threshold required for a positive spike and cannot cross zero.

Similarly, a positive output $\Delta_j^t = \mu$ cannot become negative:

$$\Delta_j^t - 2\epsilon = \mu - 2\epsilon > -\mu.$$

Hence, sign flips are impossible when $\epsilon < \mu$.

Therefore, under any perturbation bounded by $\epsilon < \mu$, the sign of the outputs of the Δ-module remains invariant. This ensures that the sparsity pattern (i.e., whether a spike occurs and its polarity) is robust to small perturbations.

As a result, the output Δ_j^t under such perturbation is constrained within the following refined bounds:

$$\left(\Delta_j^t = 0\right) \vee \left((\alpha - \beta + \mu - 2\epsilon) \leq \Delta_j^t \leq -\mu\right) \vee \left(\mu \leq \Delta_j^t \leq (\beta - \alpha + \mu + 2\epsilon)\right) \quad (6.4)$$

These bounds are essential in propagating interval constraints through the SDNN and ensuring safe approximation in formal analysis.

6.4.2 Bound Propagation through the Convolution Module

To compute the bounds of the output from the convolutional operation, we analyze how a set of bounded input values $x_j^t \in X_\Delta$, obtained from the Δ-module, propagates through the convolution layer.

A convolution operation involves applying a kernel (also known as a filter) over a local region of the input. Each output neuron in the feature map is computed as a weighted sum of a local patch in the input, followed by a bias term. Let the kernel weights be denoted as w_{ij}, where i indexes the output channel and j indexes the position in the kernel.

For the purposes of abstraction, we categorize each element of the kernel into three cases:

- $w_{ij} > 0$ (positive weights)
- $w_{ij} < 0$ (negative weights)
- $w_{ij} = 0$ (zero weights, which do not contribute to output)

Based on these cases and the bounds on each input x_j^t, we can propagate the upper and lower bounds of the convolution output by applying the interval arithmetic accordingly. This forms the basis for computing sound over-approximations of the activation ranges post-convolution.

- **Case 1: All weights** $w_{ij} \geq 0$
  - **To compute the maximum output:**
    - For $x_j^t > 0$, substitute x_j^t with $\beta - \alpha + \mu$
    - For $x_j^t < 0$, substitute x_j^t with $-\mu$
  - **To compute the minimum output:**
    - For $x_j^t > 0$, substitute x_j^t with μ
    - For $x_j^t < 0$, substitute x_j^t with $\alpha - \beta - \mu$
- **Case 2: All weights** $w_{ij} < 0$
  - **To compute the maximum output:**
    - For $x_j^t > 0$, substitute x_j^t with μ
    - For $x_j^t < 0$, substitute x_j^t with $\alpha - \beta - \mu$
  - **To compute the minimum output:**
    - For $x_j^t > 0$, substitute x_j^t with $\beta - \alpha + \mu$
    - For $x_j^t < 0$, substitute x_j^t with $-\mu$
- **Case 3: Mixed weights** $w_{ij} \in \mathbb{R}$
  - For $w_{ij} \geq 0$, apply Case 1
  - For $w_{ij} < 0$, apply Case 2

In formal verification, it is essential to conservatively approximate the output range of each neuron under input uncertainty. The above rules allow us to do that for

convolutional layers using only the input bounds and the sign of each kernel element. These symbolic bounds can then be propagated further through ReLU, pooling, or fully connected layers in a modular fashion.

Such symbolic bound propagation also enables pruning unreachable output regions early, improving SMT solver efficiency and making verification of SDNNs like PilotNet more scalable.

6.4.3 Bound Propagation through the Σ-Module

We now analyze the bound propagation through the Σ-module in the convolutional block of PilotNet. Suppose the output of the convolution operation lies within the range $[p, q]$. The Σ-module accumulates the output across time steps and applies a spike-based integration mechanism based on a threshold parameter μ and weight norm $|W|$ (the sum of absolute values over each convolutional kernel weights).

Definition 6.2 (**Accumulated Range**)**.** Let Y_Σ^t denote the output of the Σ-module at time t. The accumulated range refers to the total range of possible values that Y_Σ^t can attain across t time steps, taking into account the maximum and minimum inputs from convolution and the maximum possible decay or reset due to thresholding.

Lemma 6.3 (Accumulated Range Bounds)**.** *Let $z^t \in [p, q]$ be the bounded convolution output at each time step, and let the threshold be μ. Then the output of the Σ-module at time t, denoted as Y_Σ^t, is bounded as:*

$$t \cdot p + (t-1) \cdot |W| \cdot \mu \leq Y_\Sigma^t \leq t \cdot q - (t-1) \cdot |W| \cdot \mu.$$

Proof: Base Case ($t = 1$):

For $t = 1$, the Σ-module directly accumulates the convolutional output, so:

$$p \leq Y_\Sigma^1 \leq q.$$

Thus, the inequality holds as:

$$1 \cdot p + 0 \cdot |W| \cdot \mu = p \leq Y_{\Sigma}^{1} \leq q = 1 \cdot q - 0 \cdot |W| \cdot \mu.$$

Induction Hypothesis: Assume that the inequality holds for some $t = k \geq 1$, i.e.,

$$k \cdot p + (k-1) \cdot |W| \cdot \mu \leq Y_{\Sigma}^{k} \leq k \cdot q - (k-1) \cdot |W| \cdot \mu.$$

Induction Step: We need to prove that the inequality holds for $t = k + 1$:

$$(k+1) \cdot p + k \cdot |W| \cdot \mu \leq Y_{\Sigma}^{k+1} \leq (k+1) \cdot q - k \cdot |W| \cdot \mu.$$

At $t = k + 1$, there are two possible cases:

- **Case 1: A spike occurs.** In this case, the integral resets after threshold crossing, and the new input to the Σ-module becomes $[p + |W| \cdot \mu,\ q - |W| \cdot \mu]$. Adding this to Y_{Σ}^{k} produces Y_{Σ}^{k+1}, which satisfies the bounds as shown below.

$$\begin{aligned} k \cdot p + (k-1) \cdot |W| \cdot \mu + p + |W| \cdot \mu \leq Y_{\Sigma}^{k+1} \leq\ & k \cdot q - (k-1) \cdot |W| \cdot \mu \\ & + q - |W| \cdot \mu \\ Or, Y_{\Sigma}^{k+1} \in\ & \left[(k+1) \cdot p + k \cdot |W| \cdot \mu,\ (k+1) \cdot q - k \cdot |W| \cdot \mu\right]. \end{aligned}$$

- **Case 2: No spike occurs.** The Σ-module accumulates the next convolutional output, which lies in $[p, q]$, without any thresholding. Adding this to Y_{Σ}^{k}:

$$\begin{aligned} & \left[k \cdot p + (k-1) \cdot |W| \cdot \mu + p \leq Y_{\Sigma}^{k+1} \leq k \cdot q - (k-1) \cdot |W| \cdot \mu + q\right] \\ Or, & \left[(k+1) \cdot p + (k-1) \cdot |W| \cdot \mu \leq Y_{\Sigma}^{k+1} \leq (k+1) \cdot q - (k-1) \cdot |W| \cdot \mu\right]. \end{aligned}$$

Since this interval contains the desired bound:

$$\begin{aligned}&[(k+1)\cdot p + k\cdot|W|\cdot\mu,\ (k+1)\cdot q - k\cdot|W|\cdot\mu]\\ &\subseteq [(k+1)\cdot p + (k-1)\cdot|W|\cdot\mu,\ (k+1)\cdot q - (k-1)\cdot|W|\cdot\mu]\,.\end{aligned}$$

In other words, the inductive bound still holds.

Therefore, by the principle of mathematical induction, the bounds on Y_Σ^t hold for all $t \geq 1$.

6.5 Symbolic Modeling of PilotNet

We extend the symbolic modeling of SDNN discussed in Chapter 4 to model the PilotNet archirecture along with the following SMT-based symbolic modeling to encode the convolution operation.

SMT Encoding of Convolution Operation

We formally encode the convolution operation of the PilotNet architecture into a symbolic representation suitable for SMT-based reasoning using solvers like Z3.

Consider a convolution layer with input tensor $X \in \mathbb{R}^{C_{\text{in}}\times H\times W}$, kernel weights $W \in \mathbb{R}^{C_{\text{out}}\times C_{\text{in}}\times K_h\times K_w}$, and bias vector $b \in \mathbb{R}^{C_{\text{out}}}$ where C_{in} denotes number of input channels, C_{out} denotes number of output channels, H, W denotes the height and width of the input, K_h, K_w denotes the height and width of the kernel. The symbolic convolution output at location (i, j) in output channel c $Y_{c,i,j}$ is given by:

$$Y_{c,i,j} = \sum_{c'=0}^{C_{\text{in}}-1}\sum_{m=0}^{K_h-1}\sum_{n=0}^{K_w-1} W_{c,c',m,n}\cdot X_{c',i+m,j+n} + b_c \tag{6.5}$$

To encode this expression in SMT, we proceed as follows:

- Each input value $X_{c',i+m,j+n}$ is modeled as a symbolic variable $x_{c',i+m,j+n} \in [l, u]$, where $[l, u]$ represents the interval bounds propagated from the previous layer (e.g., the Δ-module).
- Each weight $W_{c,c',m,n}$ is a constant, loaded from the trained model.
- Each output $Y_{c,i,j}$ is introduced as a symbolic expression computed using equation (6.5).

We define the SMT constraint for each output neuron as a linear arithmetic expression over its inputs:

$$\xi_6 \triangleq \left(Y_{c,i,j} = \sum_{c'} \sum_{m} \sum_{n} W_{c,c',m,n} \cdot x_{c',i+m,j+n} + b_c \right) \tag{6.6}$$

Further, to ensure correctness under perturbations, we constrain each symbolic input with its respective bound:

$$\xi_7 \triangleq (l_{c',i+m,j+n} \leq x_{c',i+m,j+n} \leq u_{c',i+m,j+n}) \tag{6.7}$$

This symbolic encoding of the convolution operation enables reasoning over all possible input configurations within their bounds, without enumerating them.
By chaining together the SMT encodings of the convolution, ReLU, Σ, and Δ modules, we construct a complete symbolic representation of the PilotNet SDNN. This enables end-to-end verification of its safety properties.

Chapter 7

An Abstraction Refinement based verification approach towards PilotNet Verification

Verifying deep neural networks (DNNs) often involves significant computational complexity. This is due to the substantial increase in constraints that must be encoded when translating such architectures into a format suitable for solvers. To mitigate this computational overhead, we introduce a heuristic that leverages abstraction-refinement principles to improve verification efficiency. Our method systematically simplifies the network while preserving soundness guarantees [34].

To illustrate our approach, consider the example network $\mathcal{N}$ depicted in Fig. 7.1(a), which comprises only weighted sum layers. Assume we are interested in verifying whether the output y satisfies a property Q, given that the input x lies within $[-2, 2]$. It is easy to see that for this network, $h_0 = h_1 = x$ and $y = 0$, with intermediate values h_0, h_1 constrained to $[-2, 2]$.

Now, consider the network $\mathcal{N}_A$ in Fig. 7.1(b), where neuron h_0 has been removed and h_1 is treated as an input neuron with range $[-2, 2]$. Since the functional

relationship between h_1 and y remains unchanged, $\mathcal{N}_A$ can produce all outputs that $\mathcal{N}$ can. Therefore, if the verification query $\langle P \wedge (h_1 \in [-2, 2]), \mathcal{N}_A, Q\rangle$ is unsatisfiable, the original query $\langle P, \mathcal{N}, Q\rangle$ must also be unsatisfiable. In this setting, $\mathcal{N}_A$ is referred to as an *abstraction* or *over-approximation* of $\mathcal{N}$.

As a concrete case, let $P = (-2 \leq x \leq 2)$ and $Q = (y \geq 5)$. Then $\langle P \wedge (-2 \leq h_1 \leq 2), \mathcal{N}_A, Q\rangle$ is clearly unsatisfiable, as y cannot exceed 4. This implies that $\langle P, \mathcal{N}, Q\rangle$ is also unsatisfiable.

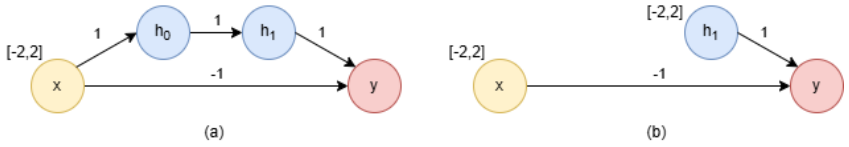

FIGURE 7.1: Illustrative example: (a) Original Network $\mathcal{N}$, (b) Abstracted Network $\mathcal{N}_A$ with neuron h_0 removed.

Formally, given a verification query $\langle P, \mathcal{N}, Q\rangle$, we aim to determine whether there exists an input x such that $P(x) \wedge Q(\mathcal{N}(x))$ holds. Let φ be a predicate, and define $\mathcal{X}_\varphi = \{x \mid \varphi(x)\}$. We construct an abstract network $\mathcal{N}_A$ that shares the same outputs as $\mathcal{N}$ but includes a superset of its input neurons. A predicate P_B is defined to bound the additional input neurons introduced in the abstraction such that $\mathcal{N}(\mathcal{X}_P) \subseteq \mathcal{N}_A(\mathcal{X}_{P \wedge P_B})$.

Lemma 7.1. *Let $\langle P, \mathcal{N}, Q\rangle$ be a verification query. Let $\mathcal{N}_A$ be a DNN with the same output neurons as $\mathcal{N}$ and an input layer that includes all input neurons of $\mathcal{N}$ along with additional ones. Suppose P_B is a constraint on the added inputs such that $\mathcal{N}(\mathcal{X}_P) \subseteq \mathcal{N}_A(\mathcal{X}_{P \wedge P_B})$. Then, if $\langle P \wedge P_B, \mathcal{N}_A, Q\rangle$ is unsatisfiable, $\langle P, \mathcal{N}, Q\rangle$ must also be unsatisfiable.*

This insight leads to an abstraction-based verification algorithm. First, solve $\langle P \wedge P_B, \mathcal{N}_A, Q\rangle$. If this is unsatisfiable, we conclude that the original query is also

unsatisfiable. Otherwise, we obtain a candidate counterexample $x \in \mathcal{X}_{P \wedge P_B}$ such that $\mathcal{N}_A(x)$ satisfies Q. We then verify whether $\mathcal{N}(x)$ satisfies Q:

- If it does, a valid counterexample is found.
- If not, the counterexample is spurious, having resulted from the abstraction.

To eliminate such spurious counterexamples, we refine the abstract network. Let $\mathcal{N}'_A$ be a refined abstraction such that its input set is more restrictive, i.e., $\mathcal{N}'_A(\mathcal{X}_{P \wedge P'_B}) \subset \mathcal{N}_A(\mathcal{X}_{P \wedge P_B})$. We then repeat the verification process on $\mathcal{N}'_A$ with updated bounds.

This refinement process continues until either:

- an unsatisfiable result is produced for some abstraction, implying the original network is also unsatisfiable, or
- a true counterexample is found that satisfies the original query.

This iterative abstraction-refinement loop is guaranteed to terminate under the assumption that the original network will eventually be restored through successive refinements. Figure 7.2 illustrates the abstraction-refinement framework.

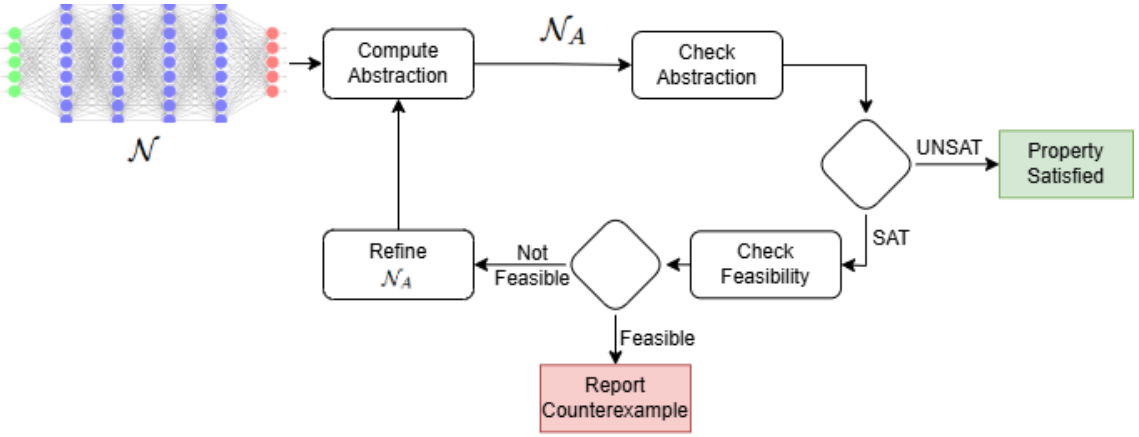

FIGURE 7.2: Abstraction-Refinement Framework for Original Network $\mathcal{N}$, Abstracted Network $\mathcal{N}_A$.

Lemma 7.2. *If the abstraction-refinement process operates over a sound and complete verification backend and terminates with the original network after finitely many refinements, then the overall framework is also sound and complete.*

The soundness ensures that false positives are not accepted, while completeness ensures that true counterexamples are eventually discovered if they exist.

The described framework is general and adaptable to different neural network architectures. However, its practical utility hinges on two key factors:

1. The method used to generate the initial abstraction.

2. The strategy employed during each refinement step.

In subsequent sections, we introduce a concrete instantiation of this framework tailored for PilotNet and demonstrate its efficacy through experimental results.

7.1 Abstraction Strategy for PilotNet

The abstraction strategy begins with identifying a subset of neurons or operations in the SDNN-PilotNet model that can be safely approximated. In the context of sigma-delta dynamics, this typically involves omitting intermediate Σ (integrator) computations and representing the resulting Δ (spike) outputs as bounded symbolic variables. These symbolic abstractions are then constrained using bounds derived from static analysis or interval propagation techniques as discussed in Chapter 6.

Once such an abstracted model $\mathcal{N}_A$ is constructed, we remove any redundant network components, such as neurons or layers that become disconnected from the output due to abstraction. The resulting abstract model is then verified against the conjunction of the original input constraints P and the abstraction-induced bounds B, forming the new query $(P \wedge B, \mathcal{N}_A, Q)$.

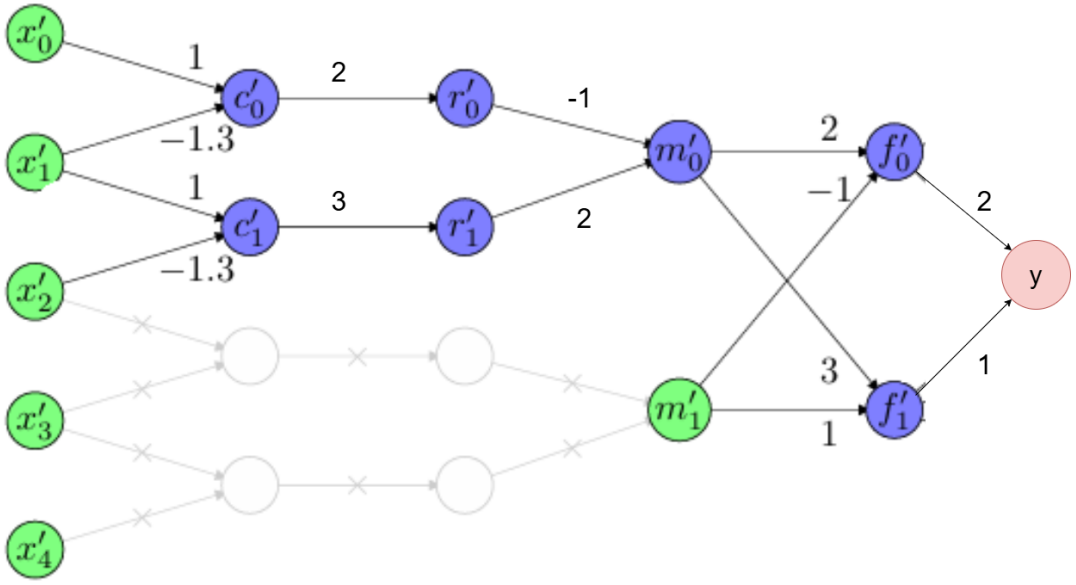

FIGURE 7.3: Abstraction Example

In the example shown in Figure 7.3, we illustrate a fragment of the SDNN-PilotNet model, where computation proceeds from the input layer (green nodes labeled x'_0 to x'_4) through a sequence of intermediate neuron layers (blue nodes), eventually producing the final output y. The network consists of several hidden layer neurons c'_i, r'_i, m'_i, f'_i. The edges are annotated with weights, assuming bias equal to zero for all layers.

To construct the abstraction, we select one or more neurons close to the output layer—specifically, the neuron m'_1 and remove all incoming edges to these neurons. This operation effectively disconnects them from the prior computation chain and treats them as symbolic variables rather than computed values. We then constrain these symbolic neurons using suitable bounds derived through static analysis techniques such as interval propagation, as discussed in Chapter 6. The resulting abstracted network, denoted $\mathcal{N}_A$, is thus a simplified version of the original network $\mathcal{N}$, where some neurons are no longer computed but abstracted.

This simplification reduces the computational burden of verification while preserving soundness by ensuring that all possible outputs of $\mathcal{N}$ are included within the behavior of $\mathcal{N}_A$. The final verification query is then posed over the abstract network using the original input constraints P along with the abstraction-induced bounds B, resulting in the query $(P \wedge B, \mathcal{N}_A, Q)$.

If this abstract query is determined to be unsatisfiable (UNSAT), we can soundly conclude that the original PilotNet satisfies the given safety property. Otherwise, if a counterexample is discovered, it must be validated on the original PilotNet. If the counterexample also violates the specification in the concrete model, we return a satisfiable result (SAT), indicating a real safety violation. However, if the counterexample is spurious—i.e., an artifact of over-approximation—we refine the abstract model by reintroducing the omitted Σ or Δ operations and reconstruct a more precise version $\mathcal{N}'_A$.

This process is repeated iteratively until a conclusive result is obtained. Due to the event-driven and sparse nature of spike-based encoding in SDNNs, and the temporal accumulation inherent to PilotNet's structure, well-chosen abstraction layers (such as the final Σ stage before output) often yield compact yet semantically faithful over-approximations. Based on our initial experiments, this significantly improves verification tractability while preserving soundness guarantees. However, not much results have scaled to obtain the desired level of efficiency when verifying more complex architectures, highlighting the need for more refined refinement heuristics.

In future work, we plan to develop structure-aware, property-driven heuristics to guide refinement, minimizing unnecessary abstraction expansion while improving counterexample discovery. Leveraging the discussed abstraction policy, we aim to evaluate this approach through experiments on the complex PilotNet architecture.

Chapter 8

Conclusion

In this dissertation, we present a symbolic SMT-based model for adversarial robustness verification of a given SDNN. Further, we show how simple bounds can be derived on the Σ units that can expedite SMT solvers. We address a novel definition of adversarial robustness well suited for temporal sequences. Through rigorous verification, we systematically analyze SDNNs' ability to handle adversarial attacks and validate our framework on temporal versions of standard datasets. In future, we can extend our framework to verify other possible definitions of adversarial robustness in the context of SDNNs. We believe our work can pave the way towards wider adoption of SDNN models going ahead. We also present a symbolic SMT-based model for adversarial robustness verification of a given SDNN based PilotNet. Additionally, we extend our SMT-based approach to verify SDNNs deployed in real-world architectures such as PilotNet. Owing to the temporal and event-driven nature of SDNNs, direct verification with vanilla SMT solvers often results in timeouts due to exponential search space growth. To address this, we introduce an abstraction-refinement framework that over-approximates the network and prunes redundant structures, significantly easing the verification burden. As future work, we plan to explore heuristics to improve the refinement procedure in PilotNet verification.

List Of Publications

- Sirshendu Das, Ansuman Banerjee, and Swarup Kumar Mohalik. 2025. Modeling and Verification of Sigma Delta Neural Networks using Satisfiability Modulo Theory. In Proceedings of the 26th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '25), June 16–17, 2025, Seoul, Republic of Korea. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3735452.3735525

Bibliography

[1] M. Bojarski *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[2] K. D. Julian *et al.*, "Deep neural network compression for aircraft collision avoidance systems," *Journal of Guidance, Control, and Dynamics*, vol. 42, no. 3, pp. 598–608, 2019.

[3] P. Prabhakar *et al.*, "Abstraction based output range analysis for neural networks," *CoRR*, vol. abs/2007.09527, 2020.

[4] A. Albarghouthi *et al.*, "Introduction to neural network verification," *Foundations and Trends® in Programming Languages*, vol. 7, no. 1–2, pp. 1–157, 2021.

[5] G. Katz *et al.*, "Reluplex: A calculus for reasoning about deep neural networks," *Formal Methods in System Design*, vol. 60, no. 1, pp. 87–116, 2022.

[6] V. Tjeng *et al.*, "Evaluating robustness of neural networks with mixed integer programming," *arXiv preprint arXiv:1711.07356*, 2017.

[7] R. Bunel *et al.*, "Branch and bound for piecewise linear neural network verification," *Journal of Machine Learning Research*, vol. 21, no. 42, pp. 1–39, 2020.

[8] K. Roy *et al.*, "Towards spike-based machine intelligence with neuromorphic computing," *Nature*, vol. 575, pp. 607 – 617, 2019.

[9] A. Pradhan *et al.*, "Model based verification of spiking neural networks in cyber physical systems," *IEEE Transactions on Computers*, vol. 72, no. 9, pp. 2426–2439, 2023.

[10] A. Gupta *et al.*, "Configuring safe spiking neural controllers for cyber-physical systems through formal verification," in *MEMOCODE 2024*, pp. 103–107, IEEE, 2024.

[11] S. Banerjee *et al.*, *SMT-Based Modeling and Verification of Spiking Neural Networks: A Case Study*, pp. 25–43. 01 2023.

[12] P. O'Connor and M. Welling, "Sigma delta quantized networks," in *International Conference on Learning Representations*, 2017.

[13] M. Bojarski *et al.*, "Explaining how a deep neural network trained with end-to-end learning steers a car," *arXiv preprint arXiv:1704.07911*, 2017.

[14] C. Szegedy *et al.*, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[15] S.-M. Moosavi-Dezfooli *et al.*, "Universal adversarial perturbations," in *CVPR 2017*, pp. 86–94, 2017.

[16] N. Akhtar *et al.*, "Threat of adversarial attacks on deep learning in computer vision: A survey," *IEEE Access*, vol. 6, pp. 14410–14430, 2018.

[17] K. Hornik *et al.*, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

[18] C. Barrett *et al.*, "Satisfiability modulo theories," in *Handbook of Satisfiability*, vol. 185 of *Frontiers in Artificial Intelligence and Applications*, ch. 26, pp. 825–885, IOS Press, Feb. 2009.

[19] H. Wu *et al.*, "Marabou 2.0: A versatile formal analyzer of neural networks," in *CAV*, pp. 249–264, July 2024. Montreal, Canada.

[20] H. Zhang *et al.*, "Efficient neural network robustness certification with general activation functions," *NeurIPS*, vol. 31, 2018.

[21] K. Xu *et al.*, "Fast and complete: Enabling complete neural network verification with rapid and massively parallel incomplete verifiers," *arXiv preprint arXiv:2011.13824*, 2020.

[22] S. Wang *et al.*, "Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification," *NeurIPS*, vol. 34, pp. 29909–29921, 2021.

[23] P. Kouvaros and A. Lomuscio, "Formal verification of cnn-based perception systems," *arXiv preprint arXiv:1811.11373*, 2018.

[24] A. Boopathy *et al.*, "Cnn-cert: An efficient framework for certifying robustness of convolutional neural networks," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 3240–3247, Jul. 2019.

[25] H.-D. Tran *et al.*, "Verification of deep convolutional neural networks using imagestars," in *International conference on computer aided verification*, pp. 18–42, Springer, 2020.

[26] Y. Wu and M. Zhang, "Tightening robustness verification of convolutional neural networks with fine-grained linear approximation," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, pp. 11674–11681, May 2021.

[27] M. Ostrovsky *et al.*, "An abstraction-refinement approach to verifying convolutional neural networks," in *International Symposium on Automated Technology for Verification and Analysis*, pp. 391–396, Springer, 2022.

[28] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks," in *Automated Technology for Verification and Analysis: 15th International Symposium, ATVA 2017, Pune, India, October 3–6, 2017, Proceedings 15*, pp. 269–286, Springer, 2017.

[29] G. Katz *et al.*, "Reluplex: An efficient smt solver for verifying deep neural networks," in *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*, pp. 97–117, Springer, 2017.

[30] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[31] G. Katz *et al.*, "The marabou framework for verification and analysis of deep neural networks," in *Computer Aided Verification*, (Cham), pp. 443–452, Springer International Publishing, 2019.

[32] L. Deng, "The MNIST database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[33] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Networks*, vol. 10, no. 9, pp. 1659–1671, 1997.

[34] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Proceedings of the 12th International Conference on Computer Aided Verification*, CAV '00, (Berlin, Heidelberg), p. 154–169, Springer-Verlag, 2000.